15-740 Final Project: Reuse-Based Adaptive Caching

Isaac Grosof and Katherine Cordwell

1 Introduction

Adaptive cache replacement policies attempt to make use of information about the executing program to achieve better cache replacement strategies. As these policies perform better and better, the question comes to mind:

What is the best possible performance of a cache replacement policy?

One attempt at answering this question was made by Belady, when he proposed his famous MIN policy. However, as this policy requires knowledge of the future, it is not a practical policy. When an implemented policy falls short of this performance, we do not know whether the gap is insurmountable, or whether a cleverer policy could do better.

Towards this end, we propose a more restricted environment, where policies only have knowledge about the future reuse distance distribution of lines of memory, rather than the actual future reuse distance, as in Belady's MIN. Many real policies use no information outside of what could be derived from the reuse distance distribution, making it a useful model.

In this restricted environment, we propose the Gittins Index policy. We derive an upper bound based on an unrealistic strengthening of the policy, which is a tighter upper bound than Belady's MIN in many circumstances. We also implement the Gittins Index policy, and find that it performs at least as well as every other practical policy tested, both on tests with a single RDD across all lines, and multiple RDDs varying between lines.

The Gittins Index policy is therefore also useful as an actual cache replacement policy, in conditions where the reuse distance distribution is known or can be cheaply measured.

2 Related Work

Adaptive cache replacement strategy share a common goal: They must gather information about the program running, and use it to decide which cache lines to evict and which to keep. Many strategies only gather information about the reuse distance distribution (RDD) of the lines in the cache. Our aim is to upper bound the possible performance of any such policy.

Some policies additionally do not differentiate between the value of different lines in the cache, allowing us to give an even better bound.

2.1 Line-unaware cache replacement

The classic LRU policy, which is widely deployed and used as a baseline is a line-unaware policy, as it does not collect any information beyond the current occupancy of a line in

the cache. All information it uses can be infered from the cache access pattern, so it is an RDD-based policy, although it does not initially appear so.

For instance, in [DZK], a protection distance strategy is chosen based on the dynamic reuse history. The dynamic reuse history is a subset of the information that is contained in the RDD, and the replacement strategy is based solely on the reuse history. As a result, the strategy in [DZK] falls squarely into our paridigm of RDD-based policies, and can be upper bounded by our methods. The policy does not collect information to differentiate different lines, but merely assigns a uniform policies accross all of them.

In [EH], data is collected about the memory access pattern, but a strategy is not provided. Specifically, the authors show how to use existing hardware structures to collect an approximation of the stack distance distribution over the execution of a program. The information on the stack distance could have been derived from the RDD, and different lines are not differentiated. The choice of data to be collected therefore shows the pervasiveness of RDD-based, line unaware models in cache replacement.

2.2 Line-aware cache replacement

Some policies are line aware, meaning that they collect data about different cache lines beyond that line's current cache occupancy, and use for more advanced caching decisions. A classic policy in this vein is the LFU policy.

In [TH], lines are classified into several access patterns, including one-touch, periodic access, and heavy access. The classification is performed based on reuse distance data, and policies are chosen for each class based on the reuse distance. This policy fits our model of an RDD-based, line aware policy.

In [SW], the SRRIP and DRRIP similarly gather information about the reuse patterns of specific cache lines, and adopt RDD-based policies with that information in mind. They frame their policy around scan and thrash resistance, trying to avoid two specific failure modes of the LRU policy. That being said, their policy fits in the model of a RDD-based line aware policy, and can be upper bounded as such.

Another paper falling into this model is [BS], which explicitly collects data on the hit age and eviction age of each cache access, and uses that information alone to perform its evictions. It is one of the most sophisticated of this class of policies, and consistently performs near our upper bound. As a result, our upper bound shows that EVA's performance is essentially as good as possible, given access only to information derivable from the RDD.

2.3 Beyond RDD

Some policies make use of information beyond what can be derived from the RDD, such as the program counter. Indeed, in real programs with multipeaked reuse distributions, the program counter is often a near perfect predictor of the true reuse distance, with which one can iimplement Belady's MIN, the true optimal policy. An example of this approach can be found in [KPK]. Note that as shown in [KPK], such policies can be extremely effective on some programs, but provide little benefit on others. We do not attempt to bound the performance of these policies at all.

3 Goals

3.1 Our Original Goals

We recall our goals. Our 99% goal was to consider very simple cases, such as the 1-line, discrete distribution case, and come up with an optimal policy or a close numerical approximation to an optimal policy. Our 75% goal was to find an optimal or very good policy for a many-line cache where each line has the same RDD. Our 50% goal was to just get a good-enough policy that consistently falls somewhere in between Belady's MIN and common RDD-based policies. Our 25% goal was to consider continuous RDDs where each cache line has a different RDD—the full-generality case—and to possibly classify some situations that were more tractable than others.

3.2 Our Progress and Modifications

We made good progress on our 99%, 75%, and 50% goals, with one notable modification: we eschewed our original plan of using PIN and running real benchmarks in favor of an abstract cache simulator, given our strong assumptions. We found that the Gittins index policy is ideal in the single-line cache situation (99% goal). Also, our experimental data indicates that the Gittins index policy consistently performs well on a multiline cache where each line has the same RDD (75% goal). We did fewer experiments with multiple RDDs, but still found that Gittins performs well on a multiline cache where the RDD of each line is drawn from some probability distribution over RDDs. Thus it seems that Gittins is the answer to our 50% goal. We did not make progress on our 25% goal, as we did not consider continuous RDDs—discrete RDDs provided more than enough interesting questions. We now describe our methods and results.

4 Gittins

The Gittins Index policy is our candidate optimal or very good RDD-based policy in our abstract setting, which we explain more in Section 5. It works by assigning an index, known as the Gittins index, to each line, and evicting the line with the smallest index whenever an eviction is needed.

The index is defined by first defining the expected value of a cache line, given a certain planned eviction time. The value is equal to the probability that the cache line will be accessed by the planned eviction time, divided by the expected time the cache line will spend in the cache until it is next accessed or evicted at the planned eviction time.

The Gittins index is defined to be the maximum expected value of a cache line over all possible planned eviction times. This can be expressed mathematically as follows:

Definition 4.1. *The Gittins index of line ℓ with reuse distribution D_ℓ and age a is*

$$\max_k \frac{Pr[D_\ell^r < k]}{E[min(D_\ell^r, k)]}$$

where $D_\ell^r = D_\ell - a | D_\ell > a$ is the remaining reuse distribution of ℓ.

4.1 Optimality

In a one line, single RDD setting, the Gittins Index policy is provably optimal. In that setting, the line will be replaced at some age, and the optimal age to do so is the age whose ratio of hit probability by that age to expected time in cache by that age is maximized.

The performance of this optimal policy is exactly the Gittins Index calculated at an initial age of 0. It is straightforward to prove that the Gittins Index is always increasing until its index of optimal value. This implies that the Gittins Index policy will achieve this optimal performance.

More generally, the Gittins Index policy achieves optimal performance if a given line can be evicted on any memory access, and if the RDD of the line it is replaced with is independent of the time step it is replaced on.

4.2 Suboptimality

In a multiline, single RDD setting, the Gittins Index is not necessarily optimal, because the policy cannot necessarily immediately evict a line at the time of its planned eviction. This is the case because multiple lines may come up for eviction on the same access, and only one can actually be evicted. In addition, a line may be evicted prematurely because of the requirement that some line must be evicted.

However, it is important to realise that neither of these problems cause a large loss in performance. These unexpected evictions happen only rarely. Moreover, they can only cause the policy to behave incorrectly if two lines have close to identical Gittins Indexes, and the amount of loss can only be as much as the difference between the true values of the lines. Therefore, there is good reason to believe that the Gittins Index policy is close to optimal on any RDD.

In a multiple RDD setting, there are greater reasons to doubt the Gittins Index policy. The Gittins Index policy does not take into account that on a hit, a line is refreshed to its original RDD, while on an eviction a line is replaced by a newly sampled line. The Gittins Index policy does not consider the original RDD of a line, but only the remaining RDD.

4.3 Upper bound

Despite the possible suboptimality of the Gittins Index policy, we can construct a theoretical upper bound on the performance of any multiline, single RDD policy. We can simply allow the Gittins Index policy to evict any number of lines on each access. In this setting, the Gittins Index policy is optimal. Moreover, any actual multiline policy is an allowable policy in this setting. This makes the performance of the Gittins Index policy in this altered setting an upper bound on the performance of any RDD-based policy in the real setting.

That being said, this upper bound will be ineffective under many RDDs. It may even be worse than Belady's MIN in some cases. However, in other cases it may be quite useful.

5 Experiments

Given our assumptions, we have opted for an abstract simulation rather than one that uses PIN and runs real benchmarks (though this would be an interesting extension of this project).

We implement a multiline fully associative inclusive cache. In our simulation, there is some probability distribution $\mathcal{D}$ over discrete RDDs. When we insert a line ℓ, we must first sample $\mathcal{D}$ to decide which RDD is associated with ℓ—call it RDD_ℓ. Then, we sample RDD_ℓ in order to specify at which timestep line ℓ will hit, if it is not evicted before then. Whenever line ℓ hits, we reinsert it with a new lifetime sampled from RDD_ℓ. Our simulator can do this because it keeps track of which RDD is associated to each line.

The cache replacement policies that we have implemented are Gittins, MIN, LRU, Economic Value Added (EVA) [BS], Expected Reuse Distance (ERD), SRRIP-FP, SRRIP-HP, BRRIP, and a static protection distance policy that we call optPD. In optPD, we iterate over many possible static protection distances (from PD = 1 to PD = the largest time in any of the RDDs) and pick the best one. This is inspired by [DZK]. In SRRIP-FP, SRRIP-HP, and BRRIP, we follow [JTSE], testing values of M (which governs the length of the RRPV) between 1 (NRU) and 5. We then pick the best one.

5.1 Experimental Results for One RDD

First, we describe some of our results in the case where all lines have the same RDD.

5.1.1 Representative Data

We show hit data after running experiments with a single, "randomly generated" RDD (also pictured) in the two plots in Figure 1. This RDD hits at time 19 with probability $\approx 74\%$ and time 37 with probability $\approx 26\%$. (Our RDDs are not perfectly random because we have some constraint on the timesteps. For example, for the plot below, we have assumed the first timestep is between 1 and 20 (since we have a 20-line cache), and each subsequent timestep is within 100 of the previous timestep. Also, we always constrain our RDDs to have between 2 and 5 peaks.)

Each vertical slice in each plots corresponds to data for a run of the simulator on x*10000 timesteps, for $x = 1$ to $x = 10$. The vertical slices are independent, i.e. correspond to different runs of the program. There is some slight variation due to the inherent variation in sampling. We normalize the hit counts to Gittins.

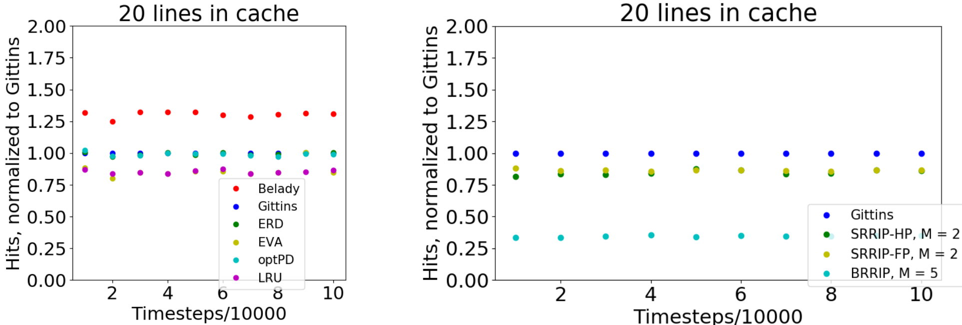

Figure 1: Representative Data for One RDD

We see that many policies are very tightly clustered around Gittins, with some exceptions:

Belady is unrealistically good, and LRU, SRRIP-FP, SRRIP-HP, and BRRIP aren't as good. LRU is highly dependent on cache size. It isn't necessarily fair to compare SRRIP-FP, SRRIP-HP, and BRRIP to Gittins in the case where all lines have the same RDD, because in this case these policies aren't able to learn any information about the RDDs. SRRIP-FP, SRRIP-HP, and BRRIP are in some sense designed to learn which lines have good RDDs, and to keep those lines—and so it makes more sense to test them in the setting where there are multiple RDDs. (Actually, in many examples with a single RDD, LRU, SRRIP-FP, SRRIP-HP, and BRRIP will perform far worse than they do here. For example, if the cache is too small, then LRU will get no hits.)

5.1.2 ERD, EVA Behave Poorly

There are some examples where ERD and EVA perform poorly. Consider the RDD where we hit at time 13 with probability 20%, at time 18 with probability 26%, and at time 115 with probability 54%. Hit data for this RDD, normalized to Gittins, is shown in Figure 2.

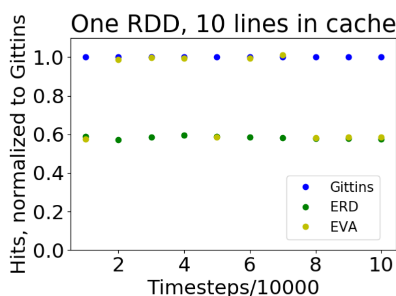

Figure 2: ERD, EVA Perform Poorly

Here, ERD consistently performs poorly. To see why, consider the case where we have two lines. Line 1 has age 13, and line 2 has age 1. Then since $ERD(13) = ((18-13) \cdot .26 + (115-13) \cdot .54)/.80 = 70.475 > ERD(1) = (13-1) \cdot .2 + (18-1) * .26 + (115-1) \cdot .54 = 68.38$, ERD will kick out line 1 whereas Gittins would keep it. This is to say, ERD doesn't realize that it's useful to keep a line with age 13 five extra timesteps.

EVA sometimes matches Gittins, but sometimes performs poorly. It seems that EVA does not always explore the future enough to know that things will hit at time 18. On some runs of the program, it does explore the future enough, and so it does well on those runs—but not on all runs.

5.1.3 Fixing EVA

If we want to fix the inconsistent behavior of EVA on examples like the above, we can seed it with the choices that Gittins would have made for a small percentage of timesteps (say 10%). We show the performance of Gittins-seeded EVA, ERD, and Gittins on the RDD from the previous example in Figure 3. We can see that Gittins-seeded EVA now matches Gittins.

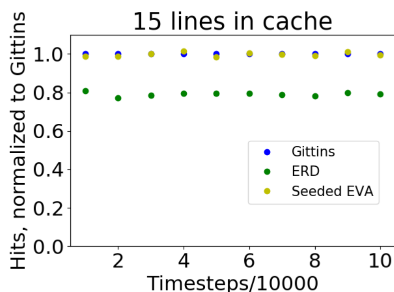

Figure 3: EVA Seeded with Gittins

5.1.4 optPD Performs Poorly

Additionally, there are rare examples where optPD performs poorly. Consider the RDD where we hit at time 9 with probability 30%, at time 20 with probability 20%, and at time 50 with probability 50%. Hit data for this RDD, normalized to Gittins, is shown in Figure 4.

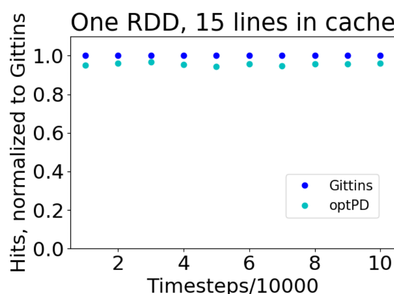

Figure 4: OptPD Performs Poorly

5.1.5 Optimal Gittins?

In the single RDD setting, we also considered the case where we allow Gittins to evict however may lines it wants (but maintaining an inclusive cache, so it has to evict at least one line) on a given step. We call this policy "optimal Gittins".

In Figure 5, we compare the performance of optimal Gittins, Gittins, and Belady on two different "randomly generated" RDDs. The first hits at time 4 with probability $\approx 25\%$, at time 35 with probability $\approx 17\%$ probability, at time 52 with $\approx 33\%$ probability, at time 63 with $\approx 13\%$ probability, and at time 185 with $\approx 12\%$ probability. The second hits at time 9 with probability $\approx 40\%$, time 28 with probability $\approx 19\%$, time 96 with probability $\approx 7\%$, time 102 with probability $\approx 18\%$, and time 174 with probability $\approx 16\%$.

We see that optimal Gittins performs much better than even Belady on the first RDD. This is because allowing a policy the ability to evict however many lines it wants is a very strong allowance, and so on some RDDs it will be able to outperform Belady, which is restricted to only evicting one line at a time. However, there are some RDDs where the

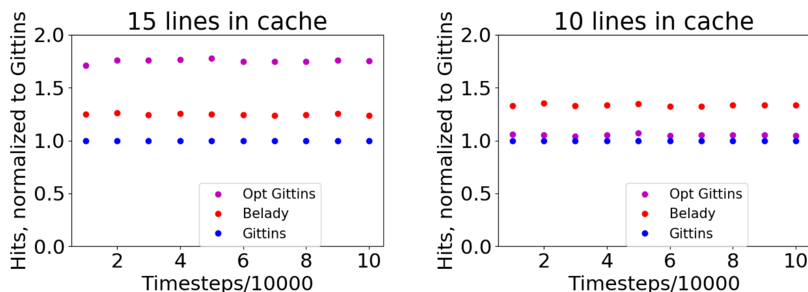

Figure 5: Optimal Gittins on Two examples

ability to evict multiple lines is not particularly useful. The second RDD is an example of this—we see in Figure 5 that the performance of optimal Gittins is close to the performance of Gittins. Strikingly, Gittins performs well even on the first RDD, where optimal Gittins is unrealistic. Its performance is shown below in Figure 6.

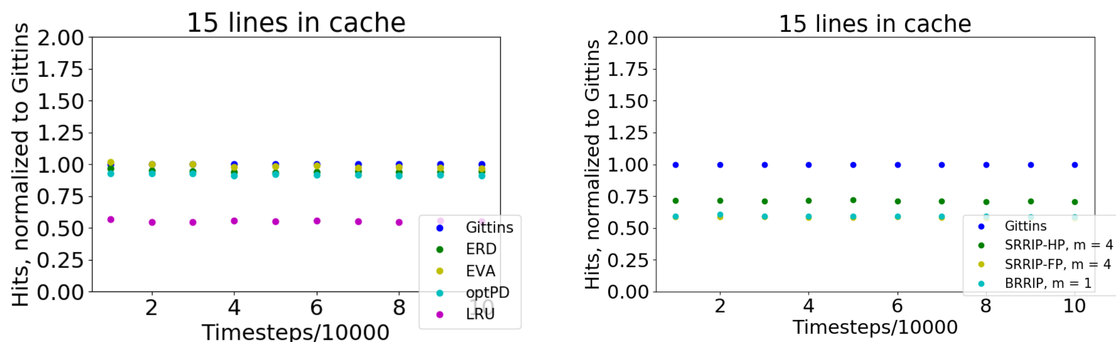

Figure 6: Gittins Performs Well on an RDD Where Optimal Gittins Dominates

5.2 Experimental Results for Multiple RDDs

Although we have admittedly performed fewer experiments with multiple RDDs, Gittins has consistently performed well on our experiments in this sphere. In Figure 7, we show hit data for three "randomly generated" RDDs. Our first RDD occurs with probability $\approx 55\%$ and hits at time 18 with probability $\approx 30\%$, at time 103 with probability $\approx 28\%$, and at time 174 with probability $\approx 42\%$. Our second RDD occurs with probability $\approx 27\%$ and hits at time 5 with probability $\approx 39\%$, at time 43 with probability $\approx 27\%$, and at time 123 with probability $\approx 34\%$. Our third RDD occurs with probability $\approx 18\%$ and hits at time 10 with probability $\approx 47\%$, at time 70 with probability $\approx 1\%$, and at time 158 with probability $\approx 54\%$.

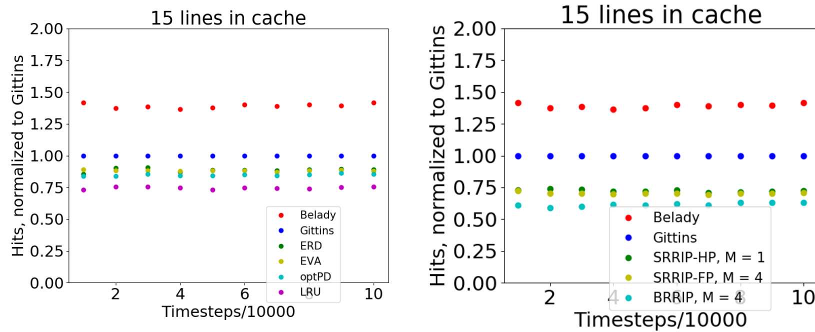

Figure 7: Multiple RDDs: Gittins Still Does Well

6 Conclusion

We find that the Gittins Index policy consistently outperforms all other RDD-based policies, while at the same time providing a tighter upper bound than Belady's MIN. All other policies tested fall short on some test cases, but Gittins performs best across all single RDD and multiple RDD tests. It is especially striking that Gittins performed so well on our test cases with multiple RDDs, given that there is less theoretical justification in that case. Future work could focus on this more.

We also find that "Optimal Gittins", which provides a theoretical upper bound on the performance of any line-unaware RDD-based policy, can provide a tighter bound than Belady's MIN in some situations.

Our main open question for the future is: Are there any conditions under which the Gittins Index policy is beaten by another RDD-based policy, whether on one line or multiple lines?

References

[BS] Nathan Beckmann and Daniel Sanchez, *Maximizing Cache Performance Under Uncertainty*, **HPCA** (2017).

[DZK] N. Duong, D. Zhao, and T. Kim, *Improving Cache Management Policies Using Dynamic Reuse Distances*, **IEEE** (2012).

[EH] D. Eklov and E. Hagersten, *Statstack: Efficient Modeling of LRU caches*, **IEEE** (2010).

[JTSE] A. Jaleel, K. Theobald, S. Steely Jr, and J. Emer, *High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)*, **ISCA** (2010).

[KPK] Keramidas, Georgios and Petoumenos, Pavlos and Kaxiras, Stefanos, *Cache replacement based on reuse-distance prediction*, **ICCD** (2007).

[SW] R. Sen and D. Wood, *Reuse-based Online Models for Caches*, **SIGMETRICS** (2013).

[TH] M. Tagaki and K. Hiraki, *Inter-Reference Gap Distribution Replacement: An Improved Replacement Algorithm for Set-Associative Caches*, **ICS** (2004).