Learning from Optimal Algorithms for Improved Software Cache Replacement

Jo Bridgwater

May 2019

Abstract
Software caches are ubiquitous in modern systems, and are often crucial for performance. Therefore, it is important to use a good cache replacement policy. However, most such caches still use the LRU heuristic, which suffers from inflexibility and a poor performance in the worst case. Computer architects have recently made significant improvements in the realm of hardware caches, so this project applies one of these strategies to software caches. There exist optimal algorithms for cache replacement, but they require knowledge of future accesses, which is impossible in practice. Since the past has predictive power over the future, Hawkeye keeps a history of cache accesses, performs the optimal policy over it, and learns that policy based on a set of features to approximate the optimal policy on future accesses. This project adds and evaluates several optimizations to adapt Hawkeye for software caches. It uses a modular approach that can be applied to any of the many types of software cache. This adapted policy produces promising results in cache hit rate without sacrificing speed.

1 Introduction

Hardware caches were originally added to processors to overcome the massive latency difference between accesses to registers and accesses to random-access memory referred to as the “memory wall” [24]. Similar “walls” in the latency of hard disk accesses were partly overcome by block caches in the operating system. Distributed systems are no different. The latency of accessing data in stores such as databases, network file systems, and content delivery networks led programmers to add local caches (usually in memory) to overcome this wall.

As early as 1985, the Sun Network Filesystem improved their base speed by about 2x largely through the addition of caches (a buffer cache, an attribute cache, and a directory cache [21]. Today, key-value stores such as Memcached [11] and Redis [17] are widely used as caches in distributed systems and easy to integrate into an existing system. For instance, Memcached is used by Quora to cache objects from their slower databases [12]. Cloud platforms such as Amazon Web Services provide similar caches to their client applications. These caches are seen as crucial contributors to the applications’ performance [22].
To achieve higher hit rates, such caching systems have been able to simply increase the capacity of the cache [20]. Unfortunately, memory is becoming more and more precious in the commodity machines that comprise most data centers. Companies are facing limits to the constant increase in size and power consumption of data centers, and taking measures to address this environmental and financial issue [2]. Memory being a major contributor to the power consumption of servers, expanding caches indefinitely is not a feasible option for the future. Even if power consumption was not a concern, hit rate only increases logarithmically with cache size [8], so the benefits of increasing cache size are limited. Therefore owners of caching systems will increasingly be looking for strategies to achieve high cache hit rates without increasing cache size.

The role of a cache replacement policy (a.k.a. cache eviction policy) is to decide which object to evict from the cache when it runs out of space. Replacement policies can significantly improve the hit rate of caches. However, compared to the sophisticated cache replacement policies recently developed for hardware caches [15, 23, 14], software systems tend to use relatively simple policies with lots of room for improvement. When I started working on this project, there were very few replacement policies for software caches even in the domain of academia.

In an effort to improve the performance of software cache replacement policies, this project uses insights from one such recent advance in hardware caching: the Hawkeye replacement policy. Hawkeye stores a history of past cache accesses in a compact form. At each cache access, it uses this history to determine what the optimal replacement policy would have done. Then, it trains a predictor on this outcome to predict which cache lines should be cached, and which should be evicted.

This project adapts the principle of Hawkeye to software caches. To do this, a few major changes and optimizations were necessary. Firstly, since there are many types of software caches with different trade-offs, this project is structured in a modular way which can be used in most of these contexts with little modification. Secondly, the core algorithm which Hawkeye uses to perform the optimal policy, though efficient for hardware caches, required some optimizations for use in software caches. Three potential optimizations are presented and evaluated: a new data structure for storing the cache history, and two relaxations of the history which slightly reduce the accuracy of the algorithm in exchange for speedups. Finally, Hawkeye’s predictor had to be adapted to learn from different features available in this new context.

I find that after applying optimizations, Hawkeye is comparable in latency to other policies, and therefore fast enough to be usable in software cache. Its hit rate is higher than LRU on the cache traces used in this evaluation, though it usually cannot do better than LHD [3], which I consider to be the state of the art in research in this area.
2 Background

2.1 Heuristic-based Cache Replacement Policies

Most modern systems use heuristic-based cache replacement policies. These policies choose victims to evict from the cache based on a static, unchanging function or a combination of them. The most well-known and widely used of these heuristics is LRU (Least Recently Used), which evicts the object that has been unused in the cache for longest. Memcached uses a version of LRU [11], and Redis allows the user to choose between 6 different heuristics based on their data [17], most of which are also variants of LRU. This relatively greater amount of choice is cited as an advantage of Redis over Memcached [13], indicating that a good replacement policy is in demand.

Heuristic approaches are simple, but suffer from an important flaw: certain reasonable access patterns draw out a degenerate worst-case performance. For instance, in LRU, when the working set is even slightly larger than the cache, and the data is accessed in a sequential pattern, the hit rate will be very low as each access causes the eviction of a cached object that is just about to be accessed. Figure 1 illustrates this access pattern.

Additionally, these heuristics’ inflexibility hurts their ability to handle even non-adversarial workloads. Redis forces the user to predict which of their six heuristics will work best on their data, which is often not forthcoming. Many prior replacement policies require significant tuning by their users to be efficient. Even if the user knows the characteristics of the access patterns the cache is likely to see, and correctly tunes the policy in accordance with them, access patterns can change over time or based on the time of day. Since heuristics work well on
some types of traces and poorly on others, the replacement policy might perform
well at some times and poorly at other times, even within a single application.
Some systems combine a few heuristics to balance out their benefits [18],
but this begs the question: instead of using flawed heuristics, why not use an
optimal policy and perform well no matter the access pattern?

2.2 Optimal Cache Replacement Algorithms

Optimal algorithms for cache replacement have the flexibility that heuristics lack. Since they make no assumptions about the access pattern of the data,
they can theoretically be used to obtain the best possible hit rate for any trace,
which can lead to significant improvements in performance.

The optimal algorithm for single-sized object cache replacement developed
by Belady [4] is quite simple: when the cache runs out of space, evict the cached
object that will be used furthest in the future. This algorithm assumes that all
objects in the cache are the same size. This is true of hardware caches, and in
a number of software cache applications, such as block caches and inode caches.
Its runtime is polynomial and relatively fast implementations exist, which makes
it an attractive option.

Even for caches with different-sized objects, there exist optimal algorithms
but the variation in cache object size upgrades the problem from polynomial-
time to NP-complete [9]. However, some effort has been put into developing
good and fast approximations which would allow close to optimal cache replace-
ment with future knowledge [6].

Unfortunately, these algorithms all require knowledge of future accesses to
the cache. They are useful for finding out how much a cache might theoretically
improve a system’s performance in the best case, given a full trace, but they
cannot be used online in real time. Nevertheless, they provide insight that can
be used to develop a realistic cache replacement policy, as this project will show.

This project focuses on the single-sized object optimal cache replacement
policy as a proof of concept, and since it most clearly illustrates the power of
the approach. We refer to it as OPT or Belady’s algorithm.

3 Hawkeye

This section contains an overview of the Hawkeye approach as laid out in previ-
ous work by Lin and Jain [14]. Then, each component of the policy is examined
more in detail followed by an explanation of my contributions which adapt it
for use in software caches.

3.1 Overview

Running optimal cache replacement algorithms is unrealistic because real sys-
tems do not know the future: the reuse distance (time until an object is accessed
again) is usually not available to the replacement policy. Luckily, past cache accesses often have strong predictive power for future accesses.

A reasonable solution might therefore be to predict the reuse distances of objects (i.e. when they will be accessed in the future), and then use these predictions to choose which object to evict. However, such approaches tend to suffer from their lack of precision: the decisions they must make are fine-grained but the predictions are only coarse-grained. For example, they might predict with high confidence that object A will be accessed again between times $t+10$ and $t+15$, but Belady’s algorithm needs to know exactly when the accesses will happen. Thus, although the predictions might be accurate, their imprecision might frequently cause them to be wrong. At best, such approaches are quite complex resulting from their need to handle this precision problem in a satisfactory way [7].

Hawkeye takes the opposite approach: first run the optimal algorithm on a history, and use past optimal eviction decisions to inform future decisions. We capture the full power of the optimal algorithm by running it on real (past) data rather than approximate (future) data, then imitating it. Learning allows us to capture higher-level insights than just reuse distance, which diminishes the problem of imprecision. Imprecise reuse distances could easily be wrong half the time, whereas the patterns learnt about the data will hold true in most cases.

3.2 Learning from OPT

Hardware Hawkeye:
What does it mean to imitate the optimal algorithm? On a high level, the goal is to learn what the optimal algorithm would have done for each access in the history, then do the same on future accesses. This can be formulated as a binary classification problem: should an accessed object be kept in the cache (cache-friendly) or should it be evicted (cache-averse)? The training data is what the optimal algorithm would have done for each access in the history.

In this project we train a perceptron positively on an access if the optimal algorithm would have cached an object, and negatively if it would not have cached it. Then, we apply the perceptron to get a prediction for the future, and store the object as cache-averse or cache-friendly. When an object needs to be evicted, Hawkeye chooses one of the cache-averse objects if there is one. If there are none, it chooses the least recently used cache-friendly object. At the beginning, the perceptron has a slight skew towards classifying objects as cache-friendly so that it has a chance to learn about them.

Software Adaptations:
In the original Hawkeye policy, the single feature used by this perceptron was the program counter of the memory access. This information is usually not available to the replacement policy of a software cache. In fact, different software cache types and implementations provide different information to the replacement policy and particular information may be useful for some caches and useless for others. For instance, some systems might know the ID of the application or pro-
gram making the cache access. In other systems, such as distributed key-value stores, this information is unlikely to be known. To handle the varying availability of information, instead of choosing a single feature, this project allows for any number of features which might be relevant to a given caching system.

Although the user can choose arbitrary features, in the evaluation for this paper, we used one feature which was available in all the evaluation traces: the key corresponding to the cached object. The key is useful for learning fine-grained information about individual cache items. In this way, it is similar to the program counter used in Hardware Hawkeye. The disadvantage is that it does not allow the perceptron to learn insights which it can apply to other objects which it has not seen before.

In this project, I tried using another feature which could be deduced from the traces available: the reuse distance itself (i.e. the time between accesses to the object). Unfortunately, I did not see any improvements in hit rate from learning from this feature. Therefore, I did not include it in the evaluation. I address the need for new features in the future work section.

3.3 OPTgen

3.3.1 Hardware OPTgen:

To know how the optimal policy would have classified an object, it is important to have an efficient way of computing the optimal. This is the purpose of OPTgen [14]. The optimal cache replacement policy for single-size caches is simply to evict the object that will be accessed again furthest into the future. Naive approaches would have excessive memory requirements and require repeatedly performing the algorithm at each time step, which is simply infeasible. Instead, OPTgen views the cache trace differently.

Each access to an object is either a hit or a miss. If it was a hit, then it must have been in the cache ever since its last access. If it was a miss, then the object was either evicted last time it was seen, or never seen before. This interval between the last access and the current access is called a liveness interval. The cost of achieving a hit on an object at a particular time is that of keeping it in the cache during its whole liveness interval. The cache trace then be represented as a set of overlapping liveness intervals (see the second level of figure 2). Each interval will either be cached or not cached, depending on the replacement policy. The replacement policy is therefore represented by the subset of liveness intervals in the trace that it chooses to cache. At any given time, the replacement policy can only choose to cache a limited number of overlapping intervals: precisely as many as fill up the size of the cache.

These overlapping intervals belonging to a replacement policy can be further simplified as a single array (see the top of figure 2). The length of the array is the total number of accesses (i.e. time steps) in the trace. Each item in the array is the size of the objects in the cache at that time, according to the replacement policy. The cache is full at times when this number is equal to the cache size. Adding an object to the cache would mean incrementing each number inside its
liveness interval by the size of the object. For instance, in figure 2, the interval labeled 1 is added to the liveness history by incrementing times 1, 2, and 3 to 1. Then the interval labeled 2 is added by incrementing [0, 4].

OPTgen uses this representation to implement OPT in the following way: when an object is reused, OPTgen considers its liveness interval. If there is space in the cache at each time during the interval, then OPTgen adds the interval to the liveness history and tells Hawkeye that OPT would have cached this interval. Because OPTgen sees the intervals in order of their reuse time, OPTgen performs the optimal policy of caching objects which are reused first, which is equivalent to evicting the object which will be accessed furthest in the future. It does so in \( O(N^2) \) time, where \( N \) is the length of the trace, because the first step of sorting the intervals can be done in a single pass, and the second step of choosing what to cache takes \( O(N^2) \) time in the worst case. The average case, however, is \( O(N * L_{avg}) \) where \( L_{avg} \) is the average length of liveness intervals.

This approach is doubly useful because it makes it possible to perform OPT over the history incrementally on each cache access. This project stores the array described above as a liveness history (shown at the top of figure 2): it represents past caching decisions. At each time step the current access is the end of a liveness interval. This liveness interval is by definition the one with the last end time of all the history so far. We can then perform a single iteration of OPTgen with it, and answer the question: would OPT have chosen to cache this object or not? This is precisely the information used to train the predictor.
3.3.2 Software Implementation of OPTgen

Even this comparatively efficient implementation of OPT is too slow to be used in some applications. The original version of OPTgen [14] is optimized for implementation in hardware. Hardware caches tend to be relatively small: on the order of a few kilobytes up to a few megabytes, with hundreds or thousands of cache lines. In contrast, software caches can be much larger since they are not restricted by being located on the processor. The original Hawkeye policy uses a history of eight times the size of the cache (or the part of the cache which is sampled). Scaling up this history to the size of software caches is very slow. As a result, OPTgen is unrealistic in software caches without optimizations. The following optimizations were considered to the end of making OPTgen usable.

**OPTgen history length**

To reduce upper bound on the overhead for each access, I considered a number of restrictions on the size of the liveness history. If the history length is 1000, then the length of OPTgen’s liveness history representation of the cache is at most 1000 and therefore at most 1000 items will need to be checked on each access. Shortening the length of the history comes with a reduction in hit rate since the information available to OPTgen is more limited. Specifically, OPTgen is able to check whether it would have been possible to cache a given object at every time within the history, but it cannot know if it was possible outside of the history. As a result, this optimization causes OPTgen to be overly optimistic in its assessment of cache friendliness.

**OPTgen granularity**

To reduce the length of the liveness history array without cutting short the trace history, we also evaluated the project using a liveness history in which one index corresponds to many time steps (see figure 3). This effectively gives the liveness history a coarser granularity. This will cause OPTgen to overestimate the cache averseness of objects because some intervals might seem to overlap even though they do not. However, it allows us to reduce the overhead of OPTgen while keeping a relatively long history.

![Figure 3: Diagram illustrating a coarser liveness history. Here, a single item in the liveness history represents two time steps (i.e. two accesses).](image)

Linked List OPTgen The last OPTgen optimization considered was to use a hybrid of a linked list and an array to represent the liveness history. The con-
sumed space in the cache does not change on every access: on many accesses, OPTgen chooses not to cache a new object, which means the consumed space at that time in the liveness history will be the exact same as the time before. Therefore, instead of storing consumed space for every single time, we can store a linked list of times when the consumed space actually changed. In the average case, this would mean iterating through fewer time steps.

However, on a cache access, a pure linked list would require sequential access to find the first node whose capacity it must check. To make this lookup time fast, we simply keep the original liveness history array, but make each item in the array a node in the linked list. This creates a linked list overlaid on top of the original liveness history, in which the nodes are changes in consumed capacity (see figure 4 for an illustration). With this hybrid data structure, finding the first node corresponding to an interval is achieved by indexing into the liveness history and stepping backwards until the first node is found. Then, OPTgen iterates through the linked list, checking the consumed capacity in each node, until the end of the interval is reached.

As a result, the complexity is determined by 1) the number of steps backward to find the first node corresponding to the desired interval and 2) the number of nodes needing to be traversed until the end of the liveness history. Although this data structure still carries an $O(N^2)$ (or $O(N)$ per access) complexity, it has the potential of being faster in the average case since it very rarely must traverse the entire liveness history, one access at a time.

4 Modularity

Most of the discussion so far has focused on the single-size object cache. However, there exist many different types of software caches. Unlike hardware caches, which are largely all single-size set-associative caches for DRAM, and whose only variation comes in the size of the cache and cache lines and the

![Figure 4: Illustration of the data structure used to optimize OPTgen. At the top, we show the original version of the liveness history as used by the basic OPTgen. A trace with 11 items only needs 6 nodes to represent it.](image-url)
amount of associativity, software caches are used for a variety of different applications, and have a variety of formats.

One important axis of variation is in the size of objects in the cache. Some caches, such as inode caches and block caches, can guarantee that all objects will be the same size. Others, such as caches in content distribution networks (CDNs), must be able to handle large and small objects. A CDN for a social network might have to cache video files of several gigabytes, as well as thumbnail images of less than a megabyte. As discussed in 2.2, different optimal algorithms and approximations exist for these cases.

Some software caches even have variable total size. In certain applications, it is useful to have a cache that increases in size, consuming more memory to achieve a better hit rate at critical times [16]. The placement of the cache in memory or on disk might also affect the trade-offs desired. Disk accesses will allow for a more complex replacement policy, whose slowness will be hidden by the higher latency of the disk access.

These factors all greatly affect the strategy of the cache replacement policy that should be applied. It would be best for cache systems to be able to adapt to the situation in which they are used, and we see promise in the Hawkeye approach being applied with modular components to achieve this adaptability. For instance, to adapt Hawkeye to a variable-size object software cache, one of the variable-size object algorithms would need to be implemented with a simple interface to this project. To adapt it to different caching systems such as Redis and Memcached, only the feature vector would need to be changed. In the future, adaptations can also be made to adapt to caches whose total size can vary.

5 Evaluation

5.1 Traces

Eviction policies are usually evaluated in simulation by measuring their hit rate on traces, which are sequences of accesses to the cache. Real-world software cache traces are not readily available in the public sphere, so the evaluation of this project is based on a combination of those real traces which have been published, and some traces which were generated to resemble real traces.

The real traces are block traces published by Microsoft Research [1]. They represent a variety of workloads from server disks at Microsoft Research [19]. Figure 5 shows the types of workloads included. Block traces are limited in their ability to represent the wide variety of caching systems. Memcached and Redis, being key-value stores, will likely differ from these traces in practice. However, these traces are used by most previous work to evaluate software caches because they are at least representative of block cache workloads, and succeed in testing cache replacement policies’ ability to deal with some important patterns in data access, such as sequential access and phase changes for example.

The other traces for evaluation were generated by the Yahoo! Cloud System
Benchmark (YCSB) [10]. YCSB is a tool for generating traces which can be used to measure the performance of a key-value store or a database. It is widely accepted as the standard in this capacity. YCSB traces are used here to supplement the real traces from MSR and to provide insight into this project’s performance on key-value traces.

YCSB is used to generate three traces, each 50M accesses long, in which the objects are accessed according to a Zipfian distribution, meaning that some objects are accessed much more frequently than others. Cache access patterns are known to often be modeled well by Zipfian distributions [8]. The traces have different percentages of read and write accesses to model different types of real-world traces.

5.2 Optimizations

The three optimizations on OPTgen described earlier were evaluated based on the latency improvements they afforded the algorithm and the performance hits they incurred.

Although the hybrid linked-list/array liveness history is equivalent to the simple array version of OPTgen and therefore does not incur any reduction in hit rate, it is clear that this optimization does not improve the latency of OPTgen measurably. Although it is true that fewer nodes are traversed on average, the data structure itself is less amenable to being cached by the processor since it is not laid out completely sequentially and its total size is larger than the simple array.

Figures 6 and 7 show the impact of the other two optimizations on latency and hit rate. Reducing the size of OPTgen’s history window improves the
latency of OPTgen. The right-hand side of figure 7 shows that the access latency of Hawkeye grows proportionally with the size of the window. The right-hand side of figure 6 shows that, similarly to the original hardware Hawkeye, the benefit of longer history tapers out after 8x the cache capacity.

Surprisingly, increasing the coarseness of the history does not harm the hit rate. In fact, it slightly beneficial for the hit rate to store history less precisely, while considerably speeding up OPTgen. This increase in hit rate might be a fluke of the traces I used. As explained in the section describing this optimization, most reuse distances are longer than 100, so it makes little difference whether there are individual slots in the history for each time step.

We therefore choose Hawkeye-8x100 for evaluation against other policies, since it has the best trade-off between latency and hit rate.

5.3 Policies

For each cache trace, the performance of several replacement policies is compared:

- Hawkeye-8x100: Hawkeye with a 8x history size, and a granularity of 100.
- LRU: least-recently used is shown to demonstrate the improvements possible over what is currently implemented in most real systems.
- LHD: is shown to represent the performance of the current best approach from academia [3].
- OPT: Belady’s algorithm, described in section 2.2 which is not possible in practice but represents the theoretical best replacement policy performance.

5.4 Hit Rate

The performance of a cache is primarily evaluated through its hit rate. Figure 8 shows the hit rates observed on a selection of traces. On these traces, it is clear that Hawkeye achieves a higher hit rate than LRU. In some cases, Hawkeye can do better than LHD, however LHD’s hit rate is often higher than Hawkeye’s. In particular, LHD performs clearly better on the YCSB-generated Zipfian traces. One possible reason for this disadvantage is that traces generated according to a distribution do not have the very eccentricities in access patterns that Hawkeye’s versatility can adapt to. In some cases, Hawkeye seems to fall back upon LRU.
This happens when OPTgen predicts objects to be more cache-friendly than they really are. Future work in improving Hawkeye’s learning module could improve its hit rate by taking into consideration features that might be more predictive.

Comparing LRU, LHD, and Hawkeye to OPT shows that there is still headroom for improvement, which can hopefully be attained with improvements on Hawkeye.

5.5 Throughput

![Graph comparing Hawkeye’s throughput to other policies](image)

Figure 9: Comparison of Hawkeye’s throughput to other policies, in accesses per millisecond.

A cache replacement policy must be fast enough that it does not bottleneck the system, since most systems are sensitive to throughput. Figure 9 compares Hawkeye-8x100 to LRU and LHD on some representative traces, showing that it can perform fast enough with optimizations to be useful in a real system. All three policies perform within an order of magnitude of each other, with several hundreds of thousands of cache accesses per second, even in an unoptimized simulator. This is fast enough to avoid being a bottleneck in a key-value cache.

6 Related Works

Least Hit Density (LHD) [3] is a state-of-the-art replacement policy for variable-sized object software caches. It uses a heuristic based on the size of objects and the predicted reuse distance to make eviction decisions. Objects are punished for being larger and rewarded for being accessed sooner.

Since work on this project began, one paper and two workshop papers were published which also attempt to push software cache replacement past the limitations of the heuristic approach. They all take into consideration a prediction of reuse distance. We take this as reinforcement of the importance of this problem and the promise of this approach.

The Optimal Steady-state Lease algorithm (OSL) [16] uses statistical methods to make predictions about future accesses. With that information, it implements a policy which can not only choose to evict objects, but can also choose
to grow or shrink the cache temporarily, if the predicted hit rate suggests that it is worthwhile.

The Prediction-error Correcting Cache [7] makes predictions about future accesses, then applies Belady’s algorithm using those predictions. Errors that are inherent in its predictions are handled by using a partitioned cache structure: objects for which the cache-friendliness prediction was higher-confidence are separated from those which were lower-confidence to lessen the impact of lower-confidence predictions.

Learning from OPT [5] (LFO) is the most similar technique to the one presented here. LFO also applies an optimal algorithm on a cache history and learns from it. However, their approach seems to learn offline (being still a workshop paper, the details are not yet clear), and then use a fully-trained gradient-boosting decision tree as their policy in production. Going forward, learning offline may have potential for doing more complex learning on data. However, the downside of this approach is its inability to adapt to new trace characteristics over time, and its reliance on training data that looks like the real data.

7 Conclusion

This project presents a new way to approach software cache replacement. I have applied the hardware cache replacement policy Hawkeye to software caches, made the optimizations necessary to achieve feasibility, and evaluated the project on a combination of real and generated cache traces. Although some work still needs to be done to further tune the policy and extract its full potential in performance, its versatility and modularity are already apparent. Therefore, taking inspiration from the advances of computer architects in hardware cache replacement is a worthwhile strategy for advancing software caching, despite the differences between the two contexts.

8 Future Work

The most important extension necessary for this project is to improve the learning module of Software Hawkeye. There are opportunities to use different features which are predictive of future accesses. For instance, the age of objects is promising since it is possible to deduce it from any trace, and was used in LHD [3] with some success. Hawkeye’s learning module could also be made more sophisticated, perhaps by using a neural network or a support vector machine to learn more complex patterns. Even the existing feature of the key could be represented more compactly, possibly with a one-hot vector.

Secondly, the project needs to be extended for use in variable-size object caches, which make up the majority of software caches. After some investigation, it seems that PFOO [6] could be suitable for such an endeavor. PFOO is an approximation of the variable-size object offline optimal which has promise in
being implemented incrementally in a similar way to Belady’s algorithm.

Finally, further evaluation on traces that are closer to a real software cache is important. Both the MSR block cache traces and the generated YCSB traces have their uses, but are fundamentally limited in their ability to predict real-world performance. In the future, if a company releases real traces from their software cache, these would be extremely valuable to all the researchers in this domain. Otherwise, the only way to be confident that a policy is good is by implementing it in a real system rather than a simulator.

9 Acknowledgements

Thank you to Calvin Lin and Akanksha Jain for advising me in this project, Michael "Tres" Brenan III for reviewing this document, my mother Karen Bridgewater for feeding me for a week before my defense, and my fellow Turing scholars for coming to my practice defense and giving me feedback.
References


