Accelerating Multi-Tier Cache Evaluations with Intelligent MRC Point Selection

A RESEARCH PROFICIENCY EXAM PRESENTED
BY
TYLER ESTRO
STONY BROOK UNIVERSITY
Technical Report FSL-21-01

January 2021

Abstract

Modern storage devices, with their diverse underlying technologies, offer a wide range of performance characteristics and tradeoffs. Such devices are often used to build multi-tier caching systems that are effective at balancing performance vs. cost. The space of possible configurations—including the number of tiers, per-tier capacities, and per-tier device properties—can be enormous. As a result, it may be infeasible to evaluate all possible combinations using traditional simulation approaches.

We introduce several approximation techniques that make it possible to identify good configurations quickly. Beyond leveraging sampling to reduce simulation costs, we dramatically reduce the number of cache configurations that need to be considered. In particular, we focus on selecting a small number of *key points* on miss ratio curves (MRCs) that represent the most useful cache sizes to simulate, such as points immediately following sharp cliffs. Our novel *Z-Method* point-selection algorithm employs statistical outlier detection to choose promising points robustly and efficiently, for both LRU and non-stack caching algorithms such as ARC. Quantitative experiments demonstrate that, compared to naive approaches, our technique significantly reduces (up to 84$\times$) the total number of simulations required to accurately identify the best configurations in terms of performance vs. cost.

To My Family.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The past decade has seen an explosion of new storage devices, which makes it difficult for system designers to evaluate which combinations of devices offer the optimal configuration for a given workload. Many studies have shown the benefits of combining several devices into a multi-tier or hybrid cache arrangement [9,24,26,44,48,49]. Alas, with more options to explore (such as the cache size at each level), and the ever-diversifying nature of workloads, the number of experimental choices quickly becomes intractable [17,18,36].

Since experimenting with physical devices is costly and time-consuming, simulation offers a more practical way to explore this large space and evaluate metrics such as throughput vs. cost. However, simulations are still expensive enough that it is difficult to explore a large number of configurations or to optimize live systems online. One first step is to sample a workload; approximation algorithms enable accurate simulation of cache behavior using only a fraction of the original trace [29,56,57].

After that, a well-known technique for quickly evaluating cache performance is Miss Ratio Curve (MRC) analysis [15,28,30,39,56]. For a given replacement algorithm, an MRC plots the cumulative miss ratio for workload requests as a function of cache size, as illustrated by the example MRC in Figure 1.1. This approach provides a simple analysis of hits and misses over a range of cache sizes for a single cache tier.

However, creating an MRC requires having a stream of cache references. In a multi-tier cache system, the references to level $n + 1$ are created by misses in—and write evictions and flushes from—level n; thus the MRC for $n + 1$ directly depends on the cache size chosen for level n. A naïve exploration of *multi-tier* configurations would thus require a separate simulation for each point in level n's MRC so as to identify the misses that become references at level $n + 1$, and hence to compute the level $n + 1$ MRC.

Since an MRC may contain anywhere from hundreds to millions of points (one for each potential cache size), this approach quickly becomes intractable. Instead, we must limit this exponential growth by keeping the number of points to simulate at level n to an absolute minimum, so that we need only simulate and generate a few MRCs at level $n + 1$. Thus, a crucial second step for evaluating multi-tier caches is to limit the number of simulations by intelligently selecting the cache sizes that will be evaluated at each level.

MRC shapes vary greatly across different workloads; some contain few interesting features, while others have many. However, it turns out to be challenging to identify points (cache sizes) at level n that are worth simulating so that the accesses that they generate can be evaluated at the next level. A simple baseline—which does not even require an MRC—is to select a fixed number k of points at each level, spacing them evenly across the entire working-set size. However, an examination of Figure 1.1 reveals that such an approach risks missing options that a designer would consider useful to explore. For example, it would probably be unwise to choose a cache size corresponding to point B in the figure when a small increase, leading to point D, would dramatically reduce the miss rate. If we had chosen $k = 20$ evenly spaced points, one of B and D would have been missed and we might conclude that a different cache size was the best we could do. In addition, we would waste resources by simulating configurations that offer little improvement (*e.g.*, even spacing might choose both points A and B in Figure 1.1).

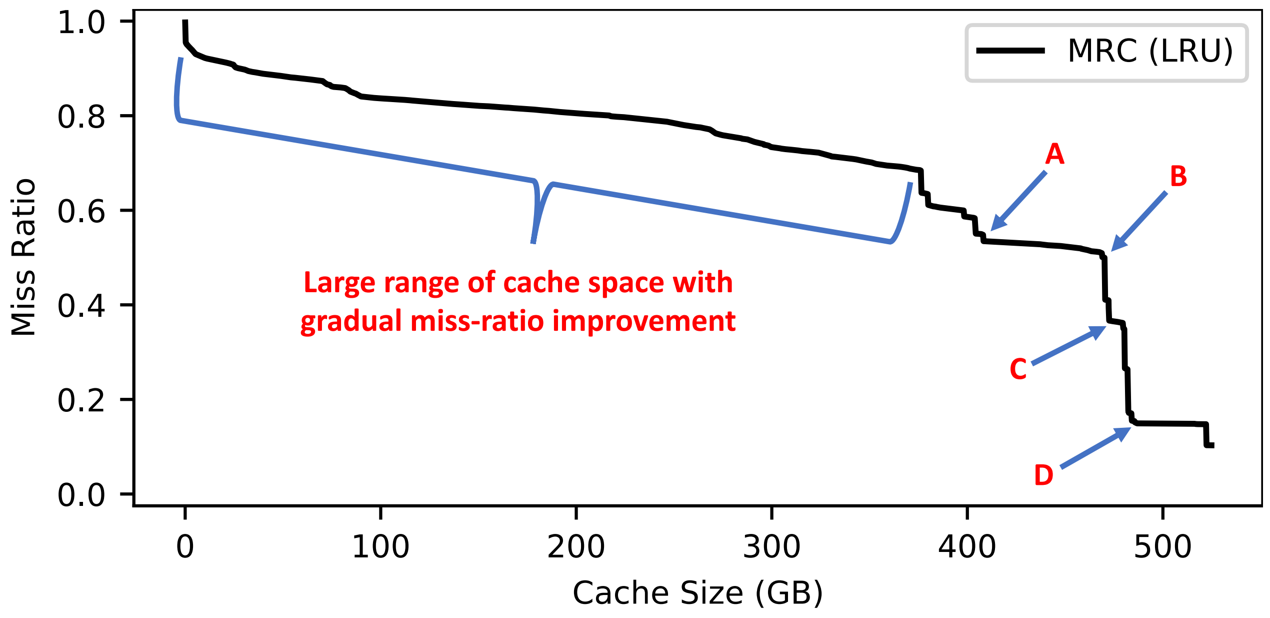

Figure 1.1: MRC for trace w10, annotated to illustrate several key points: useful "knees" (points A, C, and D), a useless "cliff" (B), and a large range of cache sizes with relatively gradual miss-ratio improvement.

Furthermore, the number of simulations grows exponentially with the number of tiers; a three-tier hierarchy would require evaluating $20^3 = 8,000$ configurations. Since each simulation takes significant time, it is critical to limit the total number of configurations investigated, which in turn suggests that it would be fruitful to choose MRC points more intelligently than with simple even spacing. Intuitively, the most promising candidates are points on an MRC where a small amount of added cache space produces a relatively large drop in the miss ratio; such points are often visible as "knees" in MRCs, as shown in Figure 1.1 (points A, C, and D).

In this paper we describe our development of a heuristic algorithm designed to pick a minimal number of *key points* in MRCs. Our new *Z-Method* algorithm finds key points efficiently and effectively, including knee points as well as other useful points to evaluate. It works for both stack (*e.g.*, LRU) and non-stack (*e.g.*, ARC) caching algorithms. To demonstrate Z-Method's usefulness at picking a small number of key points, we evaluated its efficacy on hundreds of MRCs, using it to drive a prototype multi-tier cache simulator we developed, comparing the points selected by Z-Method against the approach of choosing a fixed number of evenly-spaced points. We show that our approach can substantially reduce the number of simulated configurations needed to evaluate multi-tier caches—sometimes by more than an order of magnitude, even with only two tiers—without compromising the accurate identification of the best configurations.

This paper makes several contributions:

1. We investigated several heuristic algorithms for selecting *multiple* key points in MRCs;
2. We introduce Z-Method, an algorithm that robustly finds key points in MRCs of both stack and non-stack caching algorithms;
3. We evaluate the efficacy of Z-Method qualitatively, using hundreds of experiments, all validated by hand; and
4. We demonstrate a significant reduction in the number of multi-tier cache evaluations required to identify good configurations, when using Z-Method compared to naïve approaches.

The next section provides some background on MRCs and point-selection techniques, and describes the evolution of our heuristic algorithms, culminating with the design and evaluation of Z-Method. Section 3 applies Z-Method's point selection to accelerate the evaluation of multi-tier cache configurations, including quantitative evaluations of its performance and the quality of its results. Related work is discussed in Section 4. Finally, we summarize our conclusions and highlight opportunities for future work in Section 5.

Chapter 2

MRC Point Selection

2.1 Overview and Motivation

Many techniques have been developed to efficiently construct accurate miss ratio curves (MRCs) [25, 28, 30, 52, 53, 56, 57, 60, 65] However, more realistic simulations or physical experimentation with a specified set of cache sizes is far more complex and can be significantly more time-consuming. This complexity is compounded in multi-tier systems, where the number of possible cache size configurations grows exponentially with every added tier. In this work we sought to find the smallest number of *key points* to pick from an MRC, such that the points selected are intuitively the most useful ones to evaluate—whether via simulations or physical experiments. By doing so, we can significantly decrease the time required to adequately explore a cache configuration space.

Figure 1.1 illustrates an actual MRC for a production workload trace, annotated to highlight principles for selecting key points. For example, choosing the cache size at point C yields a miss ratio of roughly 0.4. But with just a small amount of additional cache space, one could reach point D with a 0.2 miss ratio, gaining a significant 2× reduction. Such *knees* that follow a steep *cliff* (the "L"-shaped pattern between points C and D) represent attractive key points on the MRC where a relatively small amount of additional expenditure (to buy more cache) yields large improvements in miss ratios. Similarly, point A is another key, useful point in that space. But any cache amount larger than point A and up to point B would *not* be a good point in the space: the MRC between points A and B is nearly flat—meaning that additional cache space (roughly an extra 50GB) would yield a negligible drop in miss ratio.

Knees and cliffs, however, are *not* the only patterns we have seen in MRCs. Figure 1.1 also shows a large region of cache space (more than two-thirds) where the miss ratio is not entirely flat but rather declines gradually but steadily by about 20% – a significant drop overall. There are no obvious knees for us to select. And yet, to a user configuring a cache system, who may have a limited budget or modest performance requirements, this is too large a region to ignore. An algorithm that picks useful points in the cache space should also pick *some* points along such gradually descending regions because for some workloads, even a small drop in the miss ratio can be highly desired [10, 20, 21]. The algorithm should be distribution-aware and not focus solely on knee points.

We investigated hundreds of MRCs for this work. Some exhibit simple patterns, while others are more complex. Intuitively, we want to select key points by optimizing five criteria: **(1)** Pick as few points as possible so as to reduce multi-tier simulation overheads; **(2)** First pick the knee points with the biggest "bang for the buck"—the largest reduction in miss ratio for the smallest additional investment in cache size; **(3)** Avoid any points where the MRC is essentially flat; **(4)** Identify large regions in the MRC space where there is *a gradual drop* in miss ratio, and ensure that *some* points are picked in that region; and **(5)** Minimize the overhead of the point-selection process.

To meet these five criteria, we developed several heuristic algorithms. We describe lessons learned from initial attempts that did not work well enough, and explain why they were inadequate (see Section 2.2). We continued to evolve our algorithms until we settled on one that yields near-optimal results. We call our final algorithm Z-Method (see Section 2.3).

Z-Method first computes the second derivative of an MRC at each discrete cache size using a finite-difference approximation. Relatively high, positive second-derivative values indicate a rapid change in slope, visually corresponding to a sharply decreasing line (*i.e.*, a cliff) in a curve. The subsequent second-derivative values will then decrease as the curve flattens. These properties can be used to identify knees in a curve. We then normalize the second-derivative values to generate *z-scores* (see Section 2.3.1), and focus on *outliers*. Z-Method also avoids selecting too many points that are close to each other (in both the miss-ratio and cache-size dimensions); this prevents simulating or giving users a large number of nearly identical choices. Z-Method works for stack-based algorithms (*e.g.*, LRU) as well as non-stack ones that may exhibit non-monotonicity (*e.g.*, ARC). Our algorithm also identifies knee-free regions of the MRC space where there is a near-constant slope spanning a large range of cache values, and selects a few points there. Our algorithm's computational complexity is $O(NM)$, where N is the number of points in the MRC and M is the number of selected points, which is often trivially small.

We evaluated several variants of our algorithm using 106 highly diverse block traces obtained from CloudPhysics [56]. We used both the LRU and ARC algorithms and varied our methods' hyper-parameters to demonstrate their impact on the number and quality of points selected. To the best of our knowledge, there is no known algorithm for selecting an optimal number of key points in MRCs; therefore, three members of our team *manually and independently* inspected each of hundreds of graphs produced by our techniques, to characterize the efficacy of the techniques qualitatively.

Finally, we compared the number and quality of multi-tier simulations using the points selected by our Z-Method to the method of using evenly-spaced points [62, 63]. We demonstrate this through quantitative comparisons of several latency vs. dollar-cost figures for multi-tier cache evaluations. We show that our method can explore complex multi-tier cache configuration spaces adequately using far fewer simulations than naïve approaches, reducing the amount of time required to find relevant configurations.

This work can enhance systems that analyze the performance of live installations and dynamically adjust cache properties as workload conditions change [12, 24, 32, 57]. Furthermore, Z-Method can be applied to *any* curve, and likely has many applications outside the domain of MRCs. Ultimately, this paper is a critical step towards answering an important storage-configuration question: "how much can performance be improved by spending X more dollars on cache?"

2.2 Early Attempts and Lessons Learned

Knee-detection methods Our initial goal was to find the most prominent knees in any given MRC. We first tried applying existing knee-detection algorithms, such as Kneedle [47], Dynamic First Derivative Thresholding (DFDT) [7], Dynamic Second Derivative Thresholding (DSDT) [8], and L-method [46]. To our surprise, none of the available implementations were able to find a good knee in any of our large MRCs. Upon investigation, we found that these implementations were designed for much smaller data sets, such as finding the optimal k value for k-means clustering. A typical k-means clustering analysis involves fewer than 30 points [7, 8, 37], while some of our MRCs contain several million. We confirmed with the authors of DFDT and DSDT that their implementations would require additional development to function on larger data [5].

Next, and more importantly, we learned that all of these algorithms except Kneedle were designed to find only a *single* knee [5], and could not be easily extended to find several. This was not suitable for our application, as our MRCs frequently contain multiple knees.

Finally, while Kneedle is capable of finding multiple knees in larger datasets, we contacted its authors and were unable to obtain a working implementation [3]. After careful examination of the algorithm, we decided it was also not appropriate for our application for several reasons: **(1)** Kneedle is sensitive to long tails, which are common in MRCs and present in many of our workloads; **(2)** the Kneedle algorithm applies rescaling and smoothing, which changes the shape of the curve and introduces error; **(3)** Kneedle has a time complexity of $O(N^2)$, which is not ideal for our goal of efficiently exploring many cache configurations; and **(4)** Kneedle's sensitivity parameter, S, would require significant per-curve tuning.

Thus, unable to find an appropriate technique for identifying multiple knees in a curve, we decided to develop our own.

Convex-hull techniques One of our early attempts was to construct the lower convex hull for an MRC, picking any points that intersected it. Points along the hull represent cache allocations that are inherently efficient, a property that is exploited by optimization techniques such as Talus [11] and SLIDE [57]. While this approach is guaranteed to find the most pronounced knee, it missed many of the still-valuable medium-size and smaller ones. Variant techniques that also considered points with low distances to the convex hull were similarly inadequate, selecting many useless points.

First and second derivatives Inspired by the DFDT and DSDT knee-detection algorithms, which use first and second derivatives, we then tried selecting points using derivatives. We found that picking the point in the curve with the lowest (most negative) *first* derivative would typically return a point that was in the middle of the cliff, before a knee. Conversely, taking the point with the highest *second* derivative was nearly always closer to the knee's bottom. We attributed this difference to the fact that the first derivative represents the instantaneous slope of the tangent line at any given point. This slope will always be the lowest at the steepest section of the cliff, rather than at its bottom. Conversely, the second derivative provides us with a measurement of *concavity*. As we descend along a cliff, we are in a convex region of the curve. The point where the curve flattens afterwards creates a knee, and is increasingly concave (positive second derivative)—depending on the magnitude of the cliff and the length and degree of the flat area that follows it.

Grouping Using the second derivative worked well for finding knees and quantifying their magnitude relative to others in the curve. However, this approach often produced too many points close to each knee, because, as we found, big and small knees often tend to cluster together. Therefore, we began to enforce minimum x and y distances between points as a way to select the single best point among several in a cluster of knees. This improvement picked fewer, more pronounced knees, but finding the correct x and y distance parameters for each MRC was challenging. We tried finding the ideal parameters dynamically via auto-tuning based on a desired number of points, but this resulted in an overly complex algorithm with too many hyper-parameters. We therefore continued looking for simpler solutions.

Gradually-sloped regions Another important discovery was that focusing on the knees alone sometimes resulted in no points being selected along fairly large regions of the MRC. As seen in Figure 1.1, MRCs can contain large, relatively linear areas that we still consider relevant since they cover a significant range of miss ratios. Here, the miss ratio declines gradually over a large range of cache sizes. We ultimately added an extra phase to our algorithm that scans all selected knee points, detecting regions that cover some x and y distance but contain no knees, and then selects a number of evenly-spaced points within these regions. Our technique was now even more complex, yet still sometimes missed important points.

Taking a step back, we re-focused our efforts on finding the best point associated with a knee using high positive second derivative values. We realized that once we isolated the point with the highest second

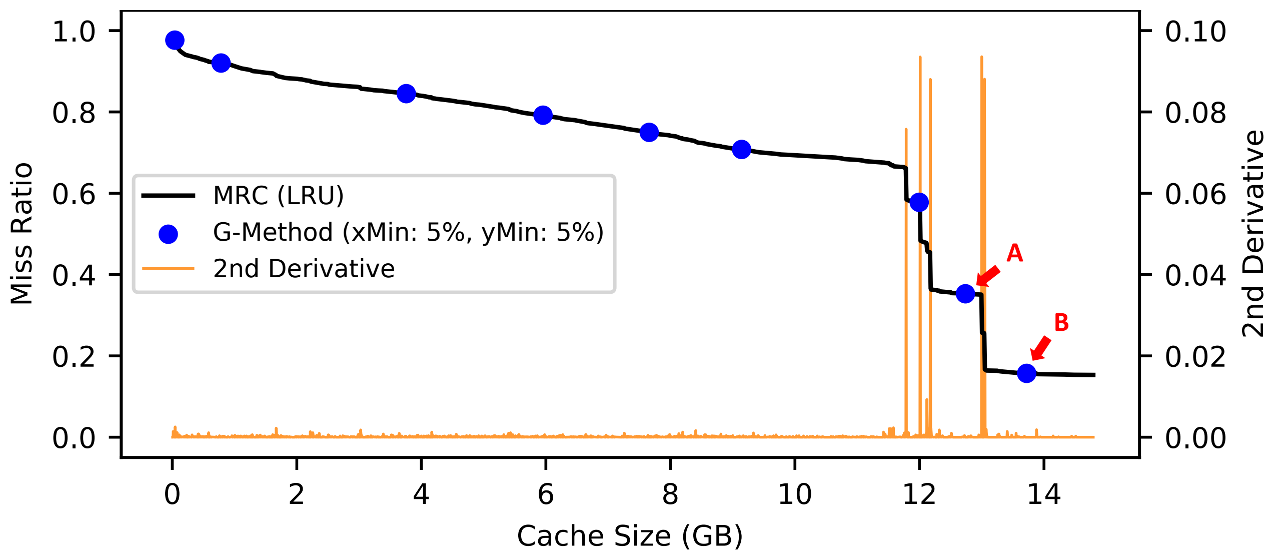

Figure 2.1: The points selected using G-Method on workload w105.

derivative, we needed to create a *window* around it and find the best point within this window. We also needed to ensure that we selected useful points in the gradually sloped sections of curves.

Grid-based G-method Our earlier experiences led us to develop a new grid-based algorithm that we call *G-Method*. G-Method finds the point with the highest second derivative, creates an $x \times y$ window surrounding it, and then picks the enclosed point with the lowest miss ratio. Any other points in the curve that have an x or y distance within the selected point's window are eliminated from future consideration. We then pick the next point in the same fashion, creating windows until the entire curve has been covered. This ensures that we will always pick the most important knees first, but that we will also eventually pick all other significant points. One can think of G-Method as progressively graying-out bands of the 2D plot in both the x and y dimensions: as each next point is selected, all other points in the gray zone are discarded. This splits a two-dimensional graph into *grids* until no more unselected points remain.

The main issue with G-Method is that it usually picks the point with the largest cache size in a given window, since that is typically the point with the lowest miss ratio in monotonically-decreasing curves. Figure 2.1 shows this: the $xMin$ and $yMin$ parameters were defined as percentages of their respective axes ranges (*e.g.*, an $xMin$ of 5% is approximately 0.75GB since the maximum cache size is 15GB). We can see that the point with the highest second derivative is near 13GB. The first window was created around 13GB, and point B was selected since it had the lowest miss ratio in the window. The next iteration created a window around 12GB, and similarly selected point A, which had the lowest miss ratio in the surrounding window. In both cases the points are suboptimal, since miss ratios that are nearly the same can be achieved by choosing points just to their left, at the bottom of each cliff.

The iterative grouping design of G-Method was simple and efficient, and did a good job at covering the entire curve, but the grouping method felt too cumbersome and ad-hoc to us. After evaluating G-Method on many different MRCs, we observed that the second derivative was usually fairly high (relative to the average) throughout an entire cliff, until it finally settled into a flat region (creating a knee).

This phenomenon can also be seen in Figure 2.1 at the two knees near 12GB and 13GB. The average second derivative in this MRC was approximately 0.00023, and there are points all throughout the width of these two cliffs with second derivatives that are two orders of magnitude greater than the average, reaching maximum values around 0.09. These findings became the foundation of our final algorithm. For G-Method, the size of our window was determined arbitrarily, based on the $xMin$ and $yMin$ parameters. By using our observation that the second derivative remains relatively high throughout the cliff, we can size these windows dynamically to avoid selecting points far away from the actual knee.

2.3 Z-Method

This section presents the design of Z-Method, our key point-selection algorithm for MRCs. This technique takes an MRC as input and selects the most useful cache sizes in the curve efficiently and robustly. It also ensures sufficient exploration of large regions in the MRC that do not contain sharp knees, but still cover a relatively significant range of cache sizes.

2.3.1 Design

Design concepts Based on our observations in Section 2.2, we found that the second derivative of an MRC is useful for dynamically identifying knees in the curve, regardless of the steepness or width of their preceding cliffs. In particular, we look for areas that have one or more points with positive second derivatives, because points that deviate greatly from the mean contain the most prominent knees.

In statistics, a *z-score* (also known as a *standard score*) is a transformation that normalizes a data value by quantifying how many standard deviations away it is from the mean [2]. Typically, a point with an absolute z-score value greater than three is considered an *outlier*. For the purpose of detecting knees in a miss ratio curve, such outliers indicate a significant change in miss ratio. The foundation of our Z-Method technique is detecting such outliers and intelligently selecting the best points among them.

We also want to minimize the number of points selected in each MRC to reduce simulation costs, by limiting the number of different cache configurations that must be simulated. To this end, we introduced two parameters to the algorithm: $xMin$ and $yMin$, which specify the minimum x and y distance between all selected points, expressed as percentages of the maximum cache size and range of miss ratios for the input MRC, respectively.

We group the points found using their second-derivative z-scores based on these parameters; doing so limits the total number of points selected, and influences the size of the selected knees. We believe it is important for a user configuring a cache to have some control over these aspects of the algorithm. For example, if they are interested only in very large knees and want to minimize the number of points, they can specify relatively high values for $xMin$ and $yMin$.

Algorithm description As seen in Algorithm 1, Z-Method takes an MRC *M* as input, along with parameters $xMin$ and $yMin$. *M* is a discrete MRC, consisting of a list of cache size and miss ratio points. $M.missRatio$ refers to all miss ratio values in *M*. We first convert the $xMin$ and $yMin$ parameters, specified as percentages, into absolute values Δx and Δy for the specified MRC (lines 1–2). This normalization ensures that these parameters function similarly for different MRCs.

We then approximate the second derivative of the MRC using the discrete central-difference formula [40], which is accomplished in linear time, producing a list of second derivatives *D*, and then discard all negative values so we do not waste time on points in the middle of cliffs (line 3). We initialize a list P that will contain all selected points, and set our starting value of $zLimit$ to 3, since a z-score ≥ 3 is a widely-accepted value for outliers (lines 4–5) [2].

We then enter a loop (lines 6–16) that selects points and progressively decrements the $zLimit$ value. First, we create a new list Z that contains candidate points: those that have a z-score greater than the current $zLimit$, and are at least a minimum Δx and Δy distance from all other already-selected points (line 7). We then group the candidate points such that each adjacent group is at least a minimum Δx distance apart from

Algorithm 1: Z-Method miss ratio curve point selection.

```
Input: Miss ratio curve $M(cacheSize_i) \mapsto missRatio_i$, $xMin$, $yMin$
Output: List of cache sizes
// convert distance percentages into absolute values for $M$
1  $\Delta x \leftarrow$ length$(M)$ $\times(xMin/100)$
2  $\Delta y \leftarrow$ (max$(M.missRatio)$ $-$ min$(M.missRatio)$) $\times(yMin$ / 100)
// compute central differences for $M$
3  $D \leftarrow$ points in $M''$ with a positive second derivative
4  Initialize empty list $P$
5  $zLimit \leftarrow 3$
6  while $TRUE$ do
7      $Z \leftarrow$ points in $D$ with a z-score $\geq zLimit$ and are at least $\Delta x$ and $\Delta y$ apart from all points in $P$
       // terminating condition
8      if $zLimit < 0$ and length$(Z) == 0$ then
           // eliminate bad points in non-monotonic curves
9          Remove points from $P$ to ensure that $missRatio$ always decreases as $cacheSize$ increases
10         return $P$
11     end
       // group points by $cacheSize$ then select the point with the lowest
          $missRatio$ from each group
12     $G \leftarrow$ points in $Z$ such that all adjacent $cacheSize < \Delta x$
13     foreach $group$ in $G$ do
14         $p \leftarrow$ point in $group$ with the lowest $missRatio$
           // enforce $\Delta y$ for new points
15         if $p$ is at least $\Delta y$ from all points in $P$ then
16             $P.append(p)$
17         end
18     end
19     $zLimit \leftarrow zLimit - 0.5$
20 end
```

other groups, enforcing the $xMin$ parameter (line 11). From each group, we select the point with the lowest miss ratio (line 13), then check that it is not within a minimum Δy distance from other points that have already been selected, enforcing the $yMin$ parameter (line 14). A point is added to the list of points P if it satisfies this constraint (line 15). We then decrement the $zLimit$ by 0.5 and continue with the next loop iteration (line 16).

This loop terminates only after we have reached a $zLimit$ less than 0 and there are no remaining points that can be selected given the $xMin$ and $yMin$ parameters (line 8). Finally, we eliminate any points that may have been poorly selected due to non-monotonicity in the MRC, which is possible with non-stack caching algorithms such as ARC [57]. We do so in a final sweep that removes points where increasing the cache size makes the miss ratio worse (line 9); clearly these points are undesirable. This simple pass requires time linear in the size of P.

2.3.2 Evaluation

We evaluated Z-Method on 106 real-world block traces collected by CloudPhysics [56], each representing week-long virtual disk activity from production VMware environments. We sampled these workloads using hash-based spatial sampling [56, 57] with a sampling rate of R=0.001 to reduce the required running time while maintaining an accurate representation of the original traces.

We present qualitative evaluations of the algorithm's parameters $xMin$ and $yMin$, as well as its overall success at finding *key points*. Furthermore, we demonstrate that Z-Method is effective for both stack and non-stack algorithms, by evaluating with LRU and ARC cache replacement policies.

Parameter: *xMin* The $xMin$ parameter has several functions within Z-Method. It is provided as a percentage of the maximum cache size in the given MRC. The most transparent effect of $xMin$ is that it constrains the minimum x distance, or cache size, between selected points. Since no two points can have an x distance less than $xMin$ between them, this provides an upper bound on the total number of selected points, and also influences the number of points that are actually selected. Because it affects the "grouping" stage of the algorithm, $xMin$ also effectively defines the width of the knees.

Figure 2.2 shows the effects of $xMin$ on workload w09 with LRU cache replacement, by fixing $yMin$ and setting $xMin$ to 1%, 5%, and 10%. The black line in each plot represents the MRC for LRU cache replacement. The green dots are the points selected by Z-Method, using the $xMin$ and $yMin$ parameters indicated in the legend. The vertical orange lines show the second derivative z-score of the MRC at each cache size. Because the z-score values have a large, workload-dependent dynamic range, we truncate them at 10 in this plot and for the remainder of the paper. A z-score range up to 10 is sufficient to identify all points considered as outliers (*e.g.*, z-score $\geq$ 3).

In Figure 2.2's MRC curve, we will focus on the knee(s) in the region of cache sizes between approximately 425GB and 475GB. In the top plot, which has $xMin = 1\%$, Z-Method considers this region to contain four separate knees, since they are at least 1% of the maximum cache size apart from each other. When we move from 1% to 5% in the middle plot, we can see that points A and B from the top plot have been removed. Those points are no longer within $xMin$ distance from each other, so they are grouped together; we are now left with two points at wider, and more prominent knees.

A similar effect is seen when we increase $xMin$ from 5% to 10% in the bottom plot. The two knees at points C and E are grouped together and C is removed. Point D is also removed, since its cache size is less than 10% away from point E. Significantly, the knee point E was favored rather than the less interesting point D.

Parameter: *yMin* The $yMin$ parameter is also specified as a relative percentage, which is then converted into an absolute value for the given MRC. It functions similarly to $xMin$, except that it constrains the y

distance, or delta miss ratio, between any two selected points. This effectively influences the height of knees and how many points are selected, while providing an upper bound on the total number of points that can be selected.

Figure 2.3 shows the effects of $yMin$ using workload w62 with LRU cache replacement by fixing $xMin$ and varying $yMin$ between 1%, 5%, and 10%. The format is otherwise the same as in Figure 2.2. In the top and the middle plots, the most interesting change occurs at point C. With $yMin$ =1% in the top plot, this very small knee is considered significant and is selected. However, when we increase $yMin$ from 1% to 5% in the middle plot, points A, B, and C are removed, as the y distance between these points and adjacent points is no longer less than $yMin$. Similarly, point E is removed when we move from 5% to 10% in the bottom plot, while the taller knee point F is retained. We can also see that point D is removed as well, as increasing $yMin$ reduces the number of selected points.

It is also important to note that for both of these parameters, we are not guaranteed to *always* have a point that is $xMin$ or $yMin$ apart from every other point. Enforcing this rule would add a great deal of complexity, and would provide little benefit, since we already select points by their order of importance.

Finding key points In Figure 2.4, we show the points selected by Z-Method with $xMin$ and $yMin$ of 5% for multiple workloads using both LRU and ARC cache replacement policies. We evaluate these plots based on whether or not they selected all of the points that we consider *key points*. To reiterate, Z-Method should first select the largest knee points and then eventually select points within any regions that cover at least 5% of the x and y axes. The first row of plots (LRU1-3) show examples where Z-Method performs well for LRU. All prominent knees are selected and large ranges of cache space with gradual decreases in miss ratio also contain an adequate number of points. The second row of plots (LRU4-6) show examples where Z-Method misses key points. For example, in plot LRU4, points A and B miss the knee points directly to their left. There are similar issues in LRU5 and LRU6.

The third row of plots (ARC1-3) show where Z-Method performs well for ARC. In addition to always selecting prominent knees and points in gradually sloped regions, we also see that points are never selected in concave regions where the miss ratio increases due to the non-monotonicity of ARC. The fourth row of plots (ARC4-6) depict where Z-Method misses key points. For example, in plot ARC4, Z-Method picks point C that is at the top of the cliff rather than the very bottom. A key feature of Z-Method is that it will never select points with a higher miss ratio that any other previously selected points with a lower cache size. This is what enables it to avoid the concave regions.

In all cases where Z-Method misses key points, the $xMin$ and/or $yMin$ parameters can be adjusted such that they would be selected. We fixed these parameters at 5% for this evaluation since we do not yet have a way of automatically selecting the ideal values. Even so, the overwhelming majority of MRCs we looked at still found all key points with these values.

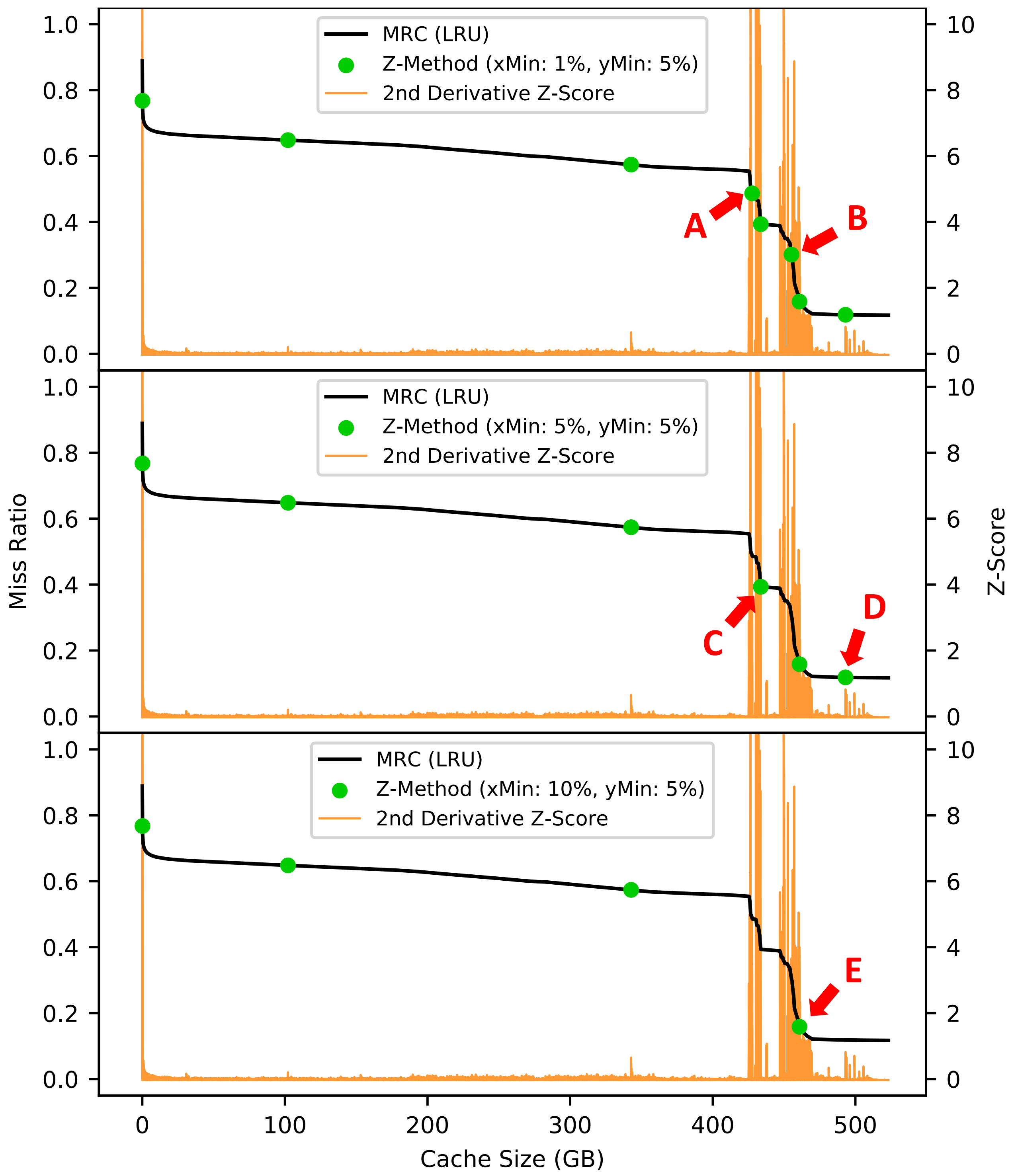

Figure 2.2: Effects of $xMin$ of 1% (top panel), 5%, and 10% on trace w09. Red arrows denote points in a panel that were not selected when the $xMin$ value increased (next panel down).

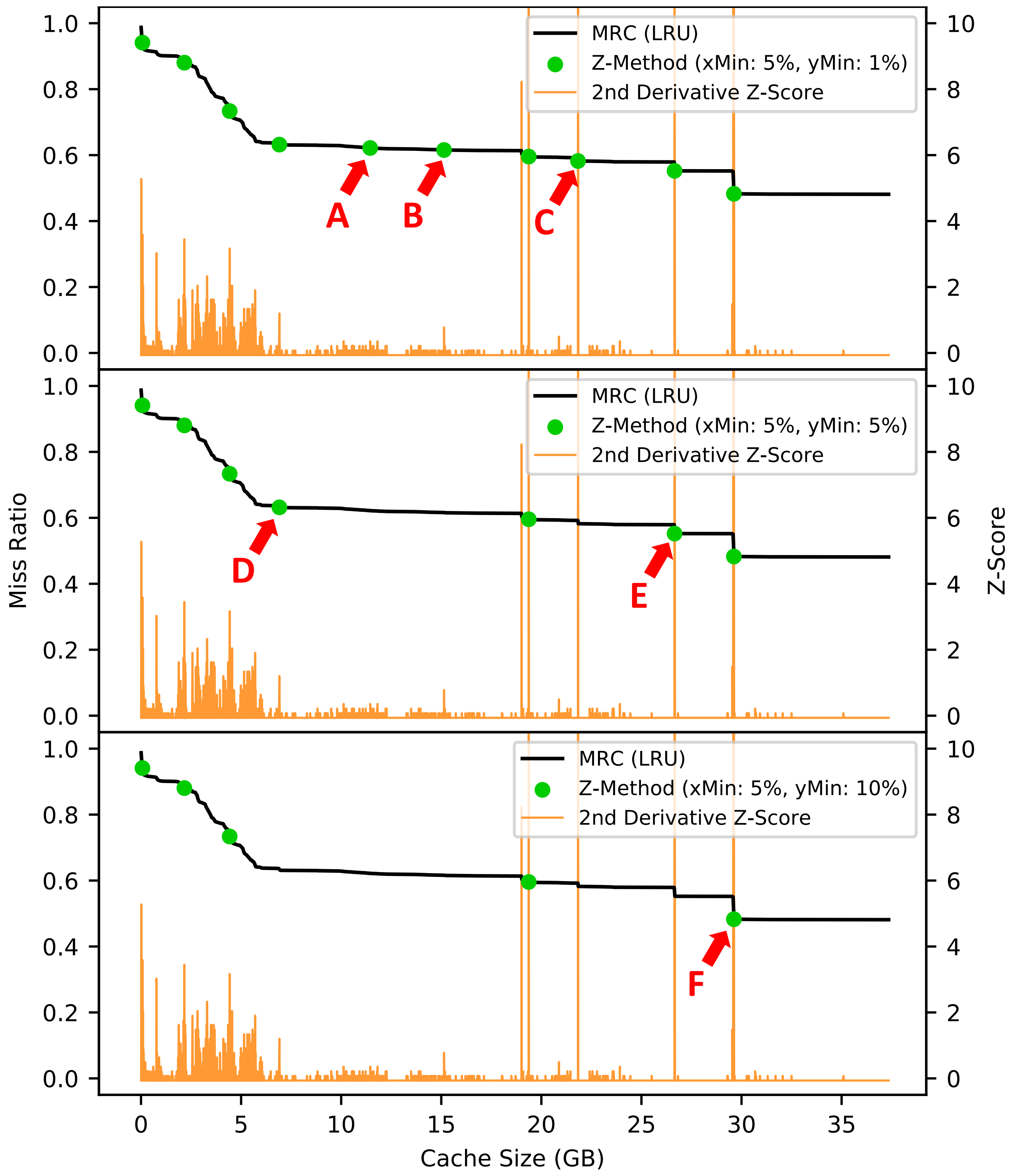

Figure 2.3: Effects of $yMin$ of 1% (top panel), 5%, and 10% on trace w62. Red arrows denote points in a panel that were not selected when the $yMin$ value increased (next panel down).

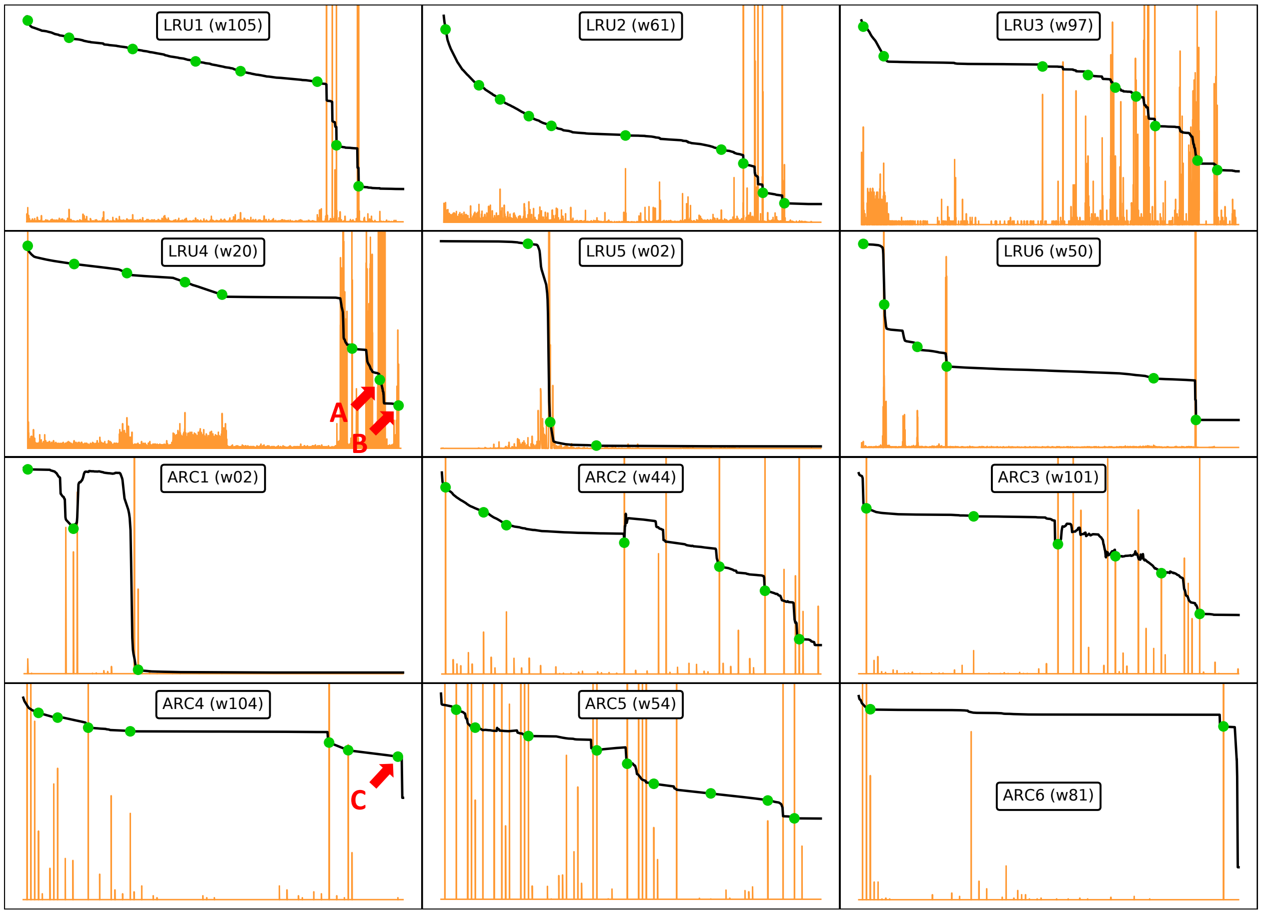

Figure 2.4: Analysis of Z-Method on LRU and ARC miss ratio curves. From the top: the first row are LRU plots where Z-Method picked fairly good points; the second row are LRU plots where Z-method missed a few, better points. The third and fourth are the same but for ARC (third row good points selected; row missed some points).

Chapter 3

Case Study: Multi-Tier Caching

The design and evaluation of multi-tier caching systems has become an incredibly complex problem. A particular system might be configured with any number and type of storage and caching devices. Interactions between tiers are governed by numerous policies that can heavily impact performance, such as write, admission, and cache replacement policies. With all these configurable parameters, efficiently finding an optimal cache configuration is a challenging task. Exploring the space through physical experimentation is costly and time-consuming; simulation allows us to evaluate configurations more quickly without having to purchase hardware. However, given such a large space, it is inefficient even to simulate every possible configuration. In this case study, we show how Z-Method can be applied to dramatically speed up the process of finding optimal cache configurations in multi-tier systems.

3.1 Design

To evaluate multi-tier systems, we extended PyMimircache [62], a storage cache simulator with an easily modifiable Python front-end and an efficient C back-end.

Our simulator has the following characteristics relevant to this work: **(1)** We implemented a global write-back policy that has two flushing mechanisms: The first is a threshold-based flush, which is designed to function similar to Linux's *vm.dirty_ratio* kernel parameter; when the fraction of dirty blocks in the cache exceeds a high threshold, we synchronously flush dirty blocks to the next tier until we reach a low threshold. We set these thresholds to 20% and 10%, respectively, similar to Linux's default values. The second mechanism is an age-based flush, based on Linux's *vm.dirty_expire_centisecs* kernel parameter. Once a block has been dirtied, it can only remain in the cache for a certain amount of time and then it is flushed. Likewise, we set this value to 30 seconds, as is the default in Linux. A limitation of our write-back policy is that we do not support asynchronous flushes. **(2)** Evicted dirty blocks are written to the next tier, and discarded if they are clean. **(3)** Tiers of DRAM are included in our simulations even though we are using block traces, which capture requests for data that was not found in DRAM. This is a limitation of the traces we are using. **(4)** In each tier, we use only the fraction of the device that matches the cache size being tested; we adjust the cost proportionately. This approach is unrealistic for real-world systems, where devices cannot be purchased in fractional increments, but is appropriate in cloud, virtual, and containerized environments that allow more fine-grained allocations and pricing. We plan to investigate such environments further in our future work. **(5)** As a means of accommodating PyMimircache, which has no concept of request size, we break all requests in our traces into 512B individual requests and process them sequentially.

Figure 3.1 depicts an overview of our PyMimircache extension. The following is a high-level outline of the simulation process: **(1)** We feed an original block I/O trace to an instance of PyMimircache; this is the top tier ("L1") of our cache hierarchy. **(2)** This instance generates two output files: (i) A log file,

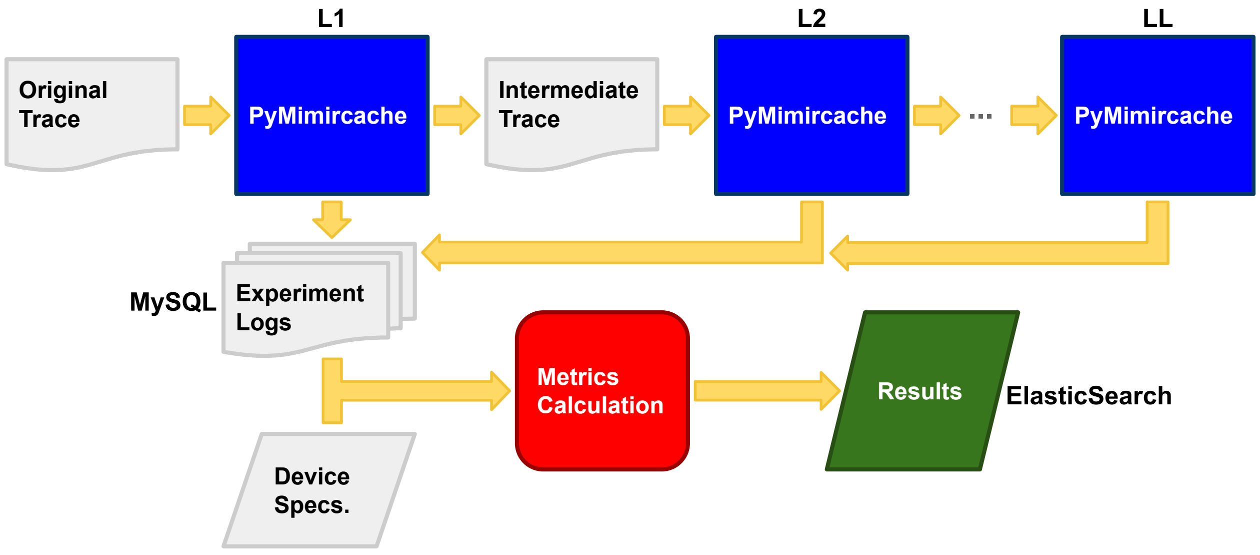

Figure 3.1: Multi-tier PyMimircache extension.

Device	Type	Price	Capacity	Avg. Read & Write Latency
Crucial DDR4 SDRAM ECC 3200	DRAM	\$105.00	16 GB	0.0138 μs, 0.0138 μs
Intel Optane SSD 905P	SSD	\$2174.00	1500 GB	10 μs, 10 μs
Seagate Enterprise Performance	HDD	\$120.17	600 GB	4,650 μs, 29,110 μs

Table 3.1: Device specifications and parameters. Prices were obtained from Amazon and Newegg in September 2020. Benchmarked specifications were correlated from device vendors, as well as independent evaluators AnandTech [4], and StorageReview [51].

"L1-log", containing counters for the following items: read hits, write hits, read misses, write misses, data read, data written, dirty evictions, clean evictions, age flushes, and threshold flushes. (ii) A new trace file, "L1-trace", which contains the read requests that missed in L1 and references to any blocks that were evicted or flushed. **(3)** After the L1 instance of PyMimircache completes, we feed the generated "L1-trace" from step 2 into another, separate instance of PyMimircache, emulating our L2 tier. **(4)** When all cache sizes have been processed, we aggregate all the log data into a single log for that experiment, then insert it into a MySQL database [41]. **(5)** We then run an analysis script with parameters that describe each tier's device purchase cost and average read and write latencies. The script calculates and records the total purchase cost and average latency for each of the experiment's cache configurations. **(6)** Finally, the metrics are inserted into an ElasticSearch database, which facilitates efficient, scalable searches and analysis [23].

3.2 Evaluation

In this section, we evaluate Z-Method's ability to speed up the process of finding optimal multi-tier cache configurations through simulations using our PyMimircache extension. We ran thousands of simulations using 106 block traces obtained from CloudPhysics [56]. Each simulation modeled an L1 cache in DRAM, an L2 cache in SSD, and a backend HDD storage device. We used the devices from Table 3.1 for all configurations, but varied their simulated sizes. We selected the cache sizes to simulate using three techniques: **(1)** Z-Method; **(2)** choosing 10 evenly-spaced cache sizes between 1 and the size of the working set (Even10); and **(3)** choosing 20 evenly-spaced cache sizes (Even20). We ran each configuration using a write-back pol-

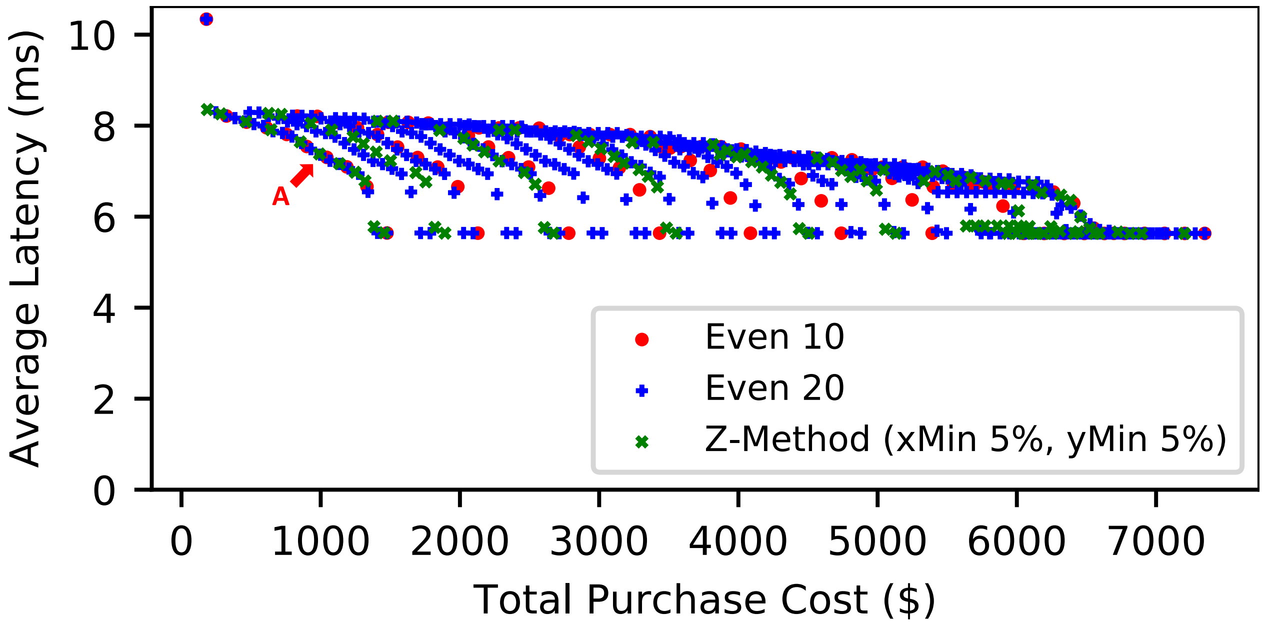

Figure 3.2: An average latency vs. total purchase cost analysis of many two-tier cache size configurations selected using 3 different point selection techniques for workload w03 using LRU policy.

icy, and used ARC or LRU cache replacement algorithms in both the L1 and L2 caches (*i.e.*, we did not evaluate configurations with a mix of LRU and ARC). The goal of these experiments was to use Z-Method to minimize the total number of cache configurations to simulate while still adequately sampling the configuration space. Thus, we compared Z-Method with Even10 and Even20 by evaluating the number of points it selects and the quality of the selected configurations.

For each simulation, we selected a combination of L1 and L2 cache sizes; the backend storage is always fixed at the size required to store all data in the workload. The combination of cache sizes and configuration parameters constituted a single multi-tier simulation that yielded a total purchase cost based on the cache and storage-device sizes, and an average latency calculated from the number of total requests served by each device.

Figure 3.2 shows the performance vs. purchase cost (lower is better) of numerous two-tier cache configurations using LRU replacement policy for workload w03. This data is critical for a user designing or reconfiguring a caching system. We selected the cache sizes using Even10, Even20, and Z-Method. Each point in the figure represents one combination of L1 and L2 cache sizes; the X axis gives the total purchase cost in dollars of the two caches plus the backend storage, while the Y axis shows the average latency (in milliseconds) of the multi-tier configuration with those cache sizes. For example, the green point A represents a simulation using approximately 178MB of DRAM as an L1 cache, 560GB of SSD as an L2 cache, and 895GB of HDD as backend storage, for a total cost of $991. The cache sizes of the simulation represented by this point were selected using Z-Method. To avoid cluttering the figure significantly, we we do not show the sizes of the individual tiers for each point.

3.2.1 Point Reduction

We now evaluate how well Z-Method can accelerate the process of finding optimal multi-tier cache configurations by minimizing the number of simulations needed to adequately explore the configuration space. Z-Method selects key points from MRCs to be simulated, as described in Section 2.3. We compare this to the naïve approach of selecting 10 or 20 evenly spaced points (Even10 or Even20) to simulate at every tier.

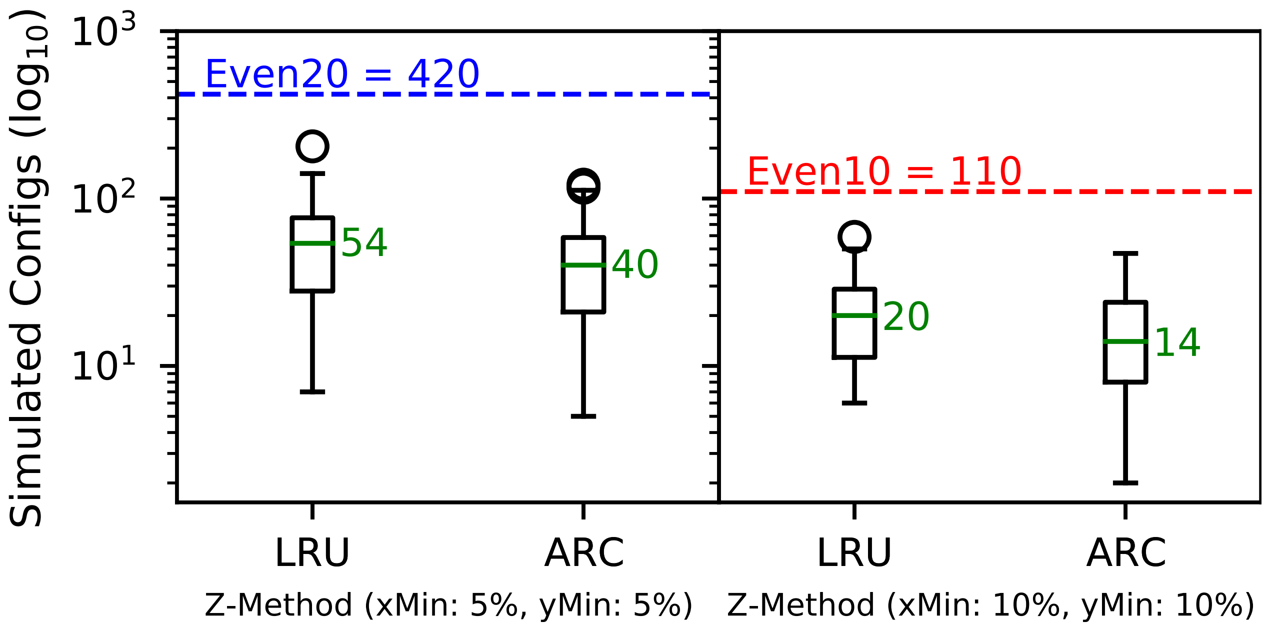

Figure 3.3: Point reduction with Z-Method. The number of two-tier cache configurations selected for simulation by Z-Method, Even10, and Even20 for each of our 106 workloads, using either LRU or ARC policies. The left panel compares Z-Method with $xMin$ and $yMin$ parameters of 5% to Even20, which always selects 420 configurations, indicated by the horizontal blue dashed line. The right panel compares Z-Method with $xMin$ and $yMin$ of 10% to Even10, which always selects 110 configurations, indicated by the horizontal red dashed line.

The goal of Z-Method is to decrease the amount of time required to adequately explore a cache configuration space. However, it is important to note that Z-Method's real-world improvements in computation time depend on several factors: the workload size, simulation software, hardware, etc. Therefore, our evaluation focuses on the more agnostic measure of the *number of points* selected for simulation, rather than timing.

The number of selected points varies based on the workload and on Z-Method's $xMin$ and $yMin$ parameters, which is an upper-bound for the total number of points that can be selected. For example, if we choose $xMin$ and $yMin$ of 5%, meaning that all selected points need to be at least 5% of the x and y axes ranges' apart from each other, then we are limited to at most 20 total points. Similarly, if we change both parameters to 10%, then we can select at most 10 points. For a fair evaluation, we use an $xMin$ and $yMin$ of 5% for comparing with Even20, and 10% values when comparing with Even10.

Figure 3.3 shows the total number of points selected for simulation by Z-Method for all 106 CloudPhysics workload traces, using both LRU and ARC cache replacement policies. Note that the figure uses a $\log_{10}$ scale on the Y axis. The plot on the left compares Z-Method with $xMin$ and $yMin$ of 5% against Even20 (the blue line), which always selects 420 points for a 2-tier configuration. Z-Method significantly reduces the number of multi-tier cache configurations that need to be simulated, achieving median reductions of 7.8$\times$ for LRU and 10.5$\times$ for ARC, compared to Even20. The right-hand plot compares Z-Method with $xMin$ and $yMin$ of 10% against Even10 (the red line), which always selects 110 points. Again, we see that Z-Method substantially reduces the number of simulated configurations, achieving median reductions of 5.5$\times$ for LRU and 7.9$\times$ for ARC.

One limitation of Z-Method is that it requires an MRC before it can select points, while picking evenly spaced points does not require a full MRC. For stack algorithms such as LRU, the MRC can be quickly calculated in one pass [39], but non-stack algorithms like ARC require much more computation. In our ex-

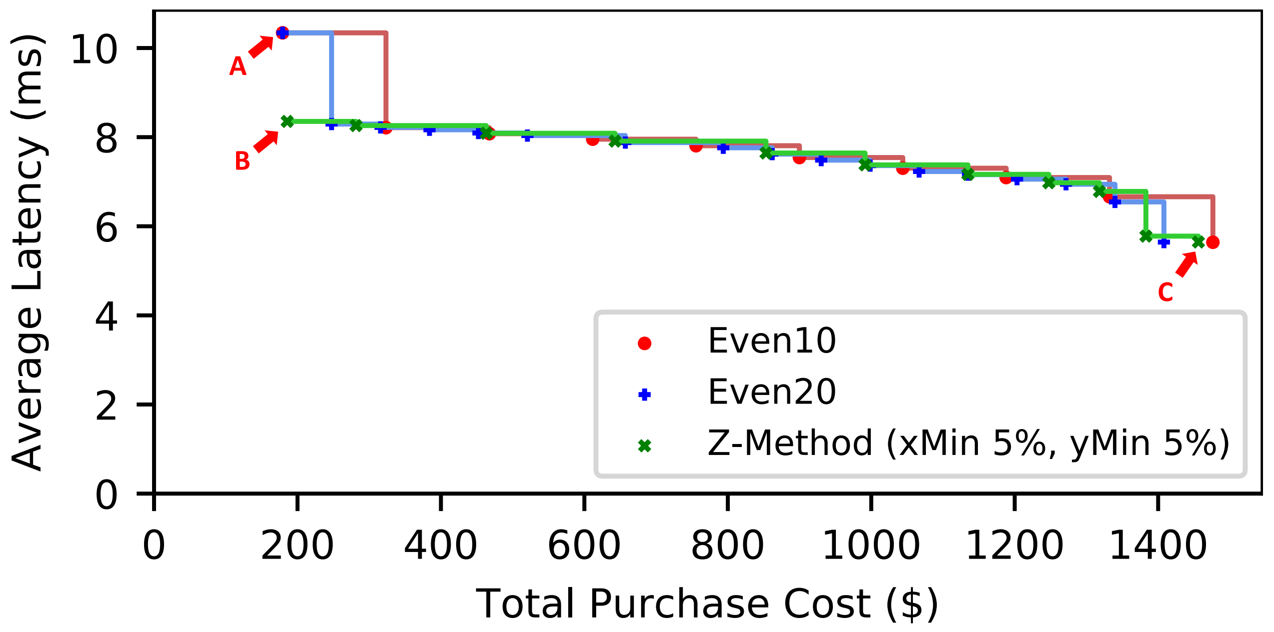

Figure 3.4: The Pareto fronts of all two-tier cache size configurations shown in Figure 3.2. The X-axis is cut off around $1500 and the figure is zoomed in around all of the Pareto optimal points, to better show the effects.

periments, we observed that Z-Method spent 30–40% of the total simulation time constructing MRCs when using the LRU policy, and between 60–70% for ARC. Even so, we *still* saw significant improvements in running time for most cases due to dramatic decreases in the required number of simulations. For example, Z-Method selected 66 points for w20 with LRU and took approximately 600 seconds to complete all simulations. Even20 selected 420 points and took around 2,800 seconds to finish, even though Even20 does not require an MRC. We also note that our experiments used only two cache tiers; the relative speedups will increase almost exponentially as we add more tiers. To illustrate this effect, we simulated w20 with 3 tiers of LRU cache. Z-Method selected only 382 points, whereas Even20 would have selected 8,420 points. Based on the 2-tier running times, we estimate that Z-Method would take around 3,400 seconds (57 minutes) to complete while Even20 would take over 56,000 seconds (15.5 hours).

3.2.2 Pareto-Optimal Configurations

Pareto optimality is a concept used to evaluate two or more solutions to a multi-objective optimization problem. In the context of multi-objective optimization, a solution is said to be *Pareto-optimal* if a given objective cannot be improved without making one or more of the other objectives worse. All the objectives are at their best possible values in a Pareto-optimal solution.

Consider the example in Figure 3.2. Here, the two objectives are average latency and total purchase cost. With an unlimited budget, one could obviously purchase enough DRAM cache to hold the entire data set, but it is rarely practical to do so. Instead, most system administrators will want to trade off cost against performance in the best possible manner, meaning that they will be interested only in Pareto-optimal solutions.

Only a small subset of all possible cache configurations are Pareto-optimal. When a set contains every Pareto-optimal configuration for a given workload and no others, it is called the *true Pareto-optimal front*. Any point in this front minimizes both the average latency and the total purchase cost; the front as a whole can be considered the “best” points.

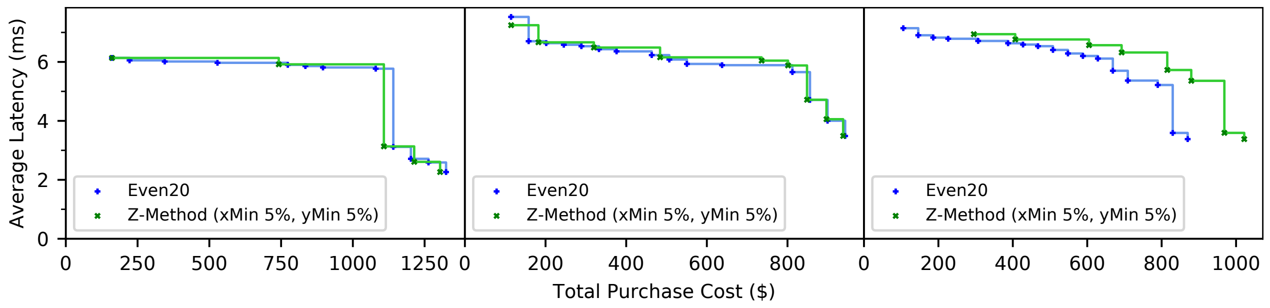

Figure 3.5: A comparison of multi-tier cache configuration Pareto fronts using cache size simulations selected by Z-Method and Even20. Each panel is classified based on how well Z-Method performs compared to Even20. The left panel depicts workload w31 and is classified as "better". The middle panel depicts workload w20 and is classified as "comparable". The right panel depicts workload w10 and is classified as "worse".

In many cases finding the true front is prohibitively expensive; a *Pareto approximation* attempts to find a set of points that are contained in or close to the true Pareto-optimal front. Since cache designers are interested only in the Pareto-optimal configurations, the task of evaluating a point-selection technique for multi-tier caching can be modeled as the evaluation of Pareto approximations.

We can more easily view the configurations that we consider interesting by taking the data from Figure 3.2 and plotting the Pareto-optimal fronts generated by each point-selection technique, as shown in Figure 3.4. We connect points using their Manhattan distance [33] to show the ranges that are dominated by both points. We also use a small threshold ϵ of the average latency range, as is commonly done in Pareto-optimal calculations, to reduce the number of selected points [34]. In our case, a point must have at least ϵ better average latency than any preceding point in the Pareto front before we select that point, because changes in average latency less than ϵ are not sufficiently interesting. The value of ϵ is subjective, but we found that 1% works well. A similar ϵ could also be applied on the X axis if desired.

Unfortunately, quantitative evaluation of Pareto approximations is a difficult and yet-unsolved problem. Many proposed techniques attempt to capture some sense of the quality of an approximation, but they are all flawed in some way, generally providing inconsistent or imprecise results. We tried several of these techniques, including the hypervolume [16], epsilon [67], and error ratio indicators [55], but the results were conflicting and uninformative.

Because of this difficulty, it is common for Pareto approximations to be evaluated using visualization [14, 54]. For example, in Figure 3.4 we can see that the green point B yields approximately 8ms average latency for around $200. Directly above point B are two (overlapping) points A that are selected by Even10 and Even20: they produce roughly 10ms average latency for the same cost of $200. Clearly, the point selected by Z-Method is superior in this case. We can also see another green point C that yields nearly the same average latency as the red circle directly to its right, but at a lower total purchase cost. All other points in this figure are very close, so we consider them equal in those ranges. From this visual inspection, we can say that in this case Z-Method yields an overall higher-quality Pareto approximation than the other two techniques.

In the absence of a robust quantitative metric, we visually inspected the Pareto fronts of all our experiments and qualified the results. In Figure 3.5, we show the Pareto fronts of Z-Method vs. Even20 and classify the quality of Z-Method into one of three classes: **(1)** The plot on the left is classified as "better"—Z-Method performs better overall than Even20; **(2)** The plot in the middle is classified as "comparable"—Z-Method performs about the same as Even20; and **(3)** the plot on the right is classified as "worse"—Z-Method performs worse overall than Even20. Our experiments yielded 424 of these plots: 106 workloads with either ARC or LRU cache replacement, and comparing Z-Method vs. Even10 or Z-Method vs. Even20. Three members of our team each independently inspected these plots and classified them into the above three classes. On average, 16% were classified as “better”, 68% were classified as “comparable”, and 16% were classified as “worse”. These results tell us that Z-Method can sometimes provide a better or worse Pareto approximation than Even10 or Even20, but is usually about as good, but with a much lower computation cost.

Chapter 4

Related Work

Simulation Modern processors are designed with multiple cores, each containing multiple tiers of cache, as well as a shared cache. Several architecture simulators have been developed and are widely used by industry and academia to facilitate engineering, research, and education [22, 31, 38, 43, 58]. One example of this is gem5, a popular architecture simulator that has been actively developed for nearly two decades [13]. It supports multiple ISAs and can accurately model complex multi-level non-uniform cache hierarchies with heterogeneous memories. Architecture simulators such as gem5 have great value, but are fundamentally different than storage simulators. Complex cache replacement algorithms that are designed specifically for storage devices (*e.g.*, SAC [19] and GCaR [61]) could not be reasonably implemented in an architecture simulator. Architecture simulators are typically driven by binaries or instruction-level traces, and could not operate on traces captured at the block, network, or system call layers.

Conversely, storage cache simulators are scarce and lacking in features. Accusim was developed to evaluate the performance impact of kernel prefetching [1]. It was designed specifically for file system caching and can not model *n* tiers. SimIdeal is a multi-tier simulator that implements several cache replacement and write policies [27]. It hard-codes the number of tiers to four and forces evictions to the immediate lower layer, and thus cannot support inclusive caching. There are also a handful of outdated simulators such as Pantheon [59]. Unfortunately, there are few storage cache simulators available, and caching research is commonly done using proprietary simulators that are not available publicly.

Knee & point selection The problem of finding knees in curves is important in many fields and has been widely studied. Angle-based method [66] uses successive differences and angles between points find knees. However, it only picks one knee point and has at least $O(N^{2.5} \log N)$ overall time complexity.

L-Method [46] fits two lines to the curve; the knee point then is their intersection. However, it again finds only one knee, and works only for curves to which lines can be fitted. Our complex MRCs make it difficult to find such lines; in addition, the computational cost of the L-Method is high. AL-Method and S-Method [6] are refinements to L-Method. AL-Method takes line angles into consideration; S-Method fits three lines to produce two knee points, so as to deal with curves that have long heads or tails. However, they share L-Method's high computational cost and cannot handle the complexity of MRCs.

The Kneedle algorithm [47] proposes a generic solution to find knees, although the authors state that a tailored solution might perform better for a specific problem. Kneedle performs several transformations on the data (notably fitting a spline to discrete points); it then looks for a place where the distance from the curve to the line $y = x$ changes in a knee-like manner. A sensitivity parameter S adjusts the behavior of the algorithm. However, Kneedle exhibits several properties that are not suitable for our needs: **(1)** Kneedle is sensitive to long tails, which are common in MRCs; **(2)** The transformations to the curve introduce error; **(3)** Its time complexity is $O(N^2)$, making it less efficient than Z-Method; and **(4)** Its S parameter would require significant per-curve tuning.

Dynamic first-derivative thresholding (DFDT) [7] finds the first-derivative values for all the points and keeps them in an array. It then calculates some threshold value using the Isodata algorithm [45]. This algorithm divides the values into high and low categories by taking an initial threshold (usually the mean). The averages of the values at or below the threshold and above are then computed. A composite average is computed using those two averages, the threshold is incremented, and the process is repeated until the threshold is larger than the composite average. A minimum distance is calculated by subtracting this threshold value from the first element in the first derivative array. Thereafter, the threshold value is subtracted from each element of the array and if the result is smaller than the minimum distance, then the value of the minimum distance is updated. The final knee point is the index of the element which gives the minimum distance of the threshold value. This algorithm has a limitation that it returns only one knee point and $O(N^2)$ time complexity.

Dynamic second derivative thresholding (DSDT) [8] is similar to DFDT except that it uses second derivatives. The remaining steps are the same: calculate the threshold value using the Isodata algorithm, take the difference of this threshold with each element in the second derivative array, then return the index that gives the lowest difference with the threshold value. Since the algorithm is similar to DFDT, its limitations are also the same.

Multi-tier caching Multi-tier caching is an active research topic in many areas, including VM management [44], heterogeneous networks [35], cloud storage [49], hardware designs [42,50], and data-centers [36, 64]. Our work is applicable in all of these areas, but they each focus on problems that are somewhat different than ours.

MultiCache [44] is a multi-layer cache-management system that dynamically partitions the cache device based on the locality and priority of workloads. To maximize overall performance, a greedy heuristic adjusts the sizes of each cache layer iteratively, which reduces latency and increases the hit ratio for VM workloads. However, MultiCache considers only the size of cache partitions, ignoring their monetary cost.

eMRC [36] is a new technique that efficiently approximates the convex hull of the multi-dimensional miss-rate surface for a multi-tier cache. This approximation is valuable in conjunction with cliff-removal techniques based on Talus [11], which eMRC generalizes to multiple tiers. However, eMRC cannot be used to model multi-tier cache systems that do not employ cliff removal. In contrast, our Z-Method approach does not require convexity to accelerate multi-tier evaluations, making it broadly applicable to production deployments of existing multi-tier caches.

GTSSL [49] uses a multi-tier compaction algorithm and advanced data structures (*e.g.*, Bloom filters) to improve performance compared to Cassandra and HBase. Our method of multi-tier cache evaluation can help systems such as GTSSL improve their hit ratios and cost-effectiveness.

Chapter 5

Conclusion

Multi-tier caching systems have a large number of possible configurations that produce a wide range of performance and costs. As the configuration space continues to grow due to advancements in caching and storage technology, exploring the entire the space through traditional simulation becomes infeasible.

In this work, we investigated several heuristic algorithms for selecting key points in MRCs to simulate, drastically reducing the cost of exploration. We introduced Z-Method, an algorithm that robustly and efficiently identifies multiple key points in MRCs with minimal overhead. We evaluated Z-Method's efficacy through hundreds of experiments using our extended version of PyMimircache [62]. We demonstrated that Z-Method can be applied to dramatically reduce the number of simulations required to identify good multi-tier cache configurations, by up to a factor of 84× when compared to naïve point-selection techniques.

Future work Z-Method has a few tunable hyper-parameters and built-in variables whose ideal values vary among workloads. We are currently investigating ways to intelligently guide their selection based on the characteristics of a workload's MRC (*e.g.*, the amount of entropy). We are also exploring whether we can improve on existing knee-detection techniques, or incorporate them into Z-Method to further enhance point selection and decrease overheads (especially for generating intermediate MRCs). Finally, since Z-Method can be used on *any* curve, we are investigating additional applications outside the realm of cache design.

Bibliography

[1] AccuSim: Accurate simulation of cache replacement algorithms, March 2020. https://engineering.purdue.edu/~ychu/accusim/.

[2] Charu C. Aggarwal. *Outlier Analysis*. Springer Publishing Company, Incorporated, 2nd edition, 2016.

[3] Jeannie R. Albrecht. Personal communication via e-mail, October 2020.

[4] Anandtech: Hardware news and tech reviews since 1997. www.anandtech.com.

[5] Mário Antunes. Personal communication via e-mail, December 2020.

[6] Mário Antunes, Henrique Aguiar, and Diogo Gomes. AL and S methods: Two extensions for L-method. In *7th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 371–376. IEEE, 2019.

[7] Mário Antunes, Diogo Gomes, and Rui L Aguiar. Knee/elbow estimation based on first derivative threshold. In *Fourth IEEE International Conference on Big Data Computing Service and Applications (BigDataService)*, pages 237–240, Bamberg, Germany, March 2018. IEEE.

[8] Mário Antunes, Joana Ribeiro, Diogo Gomes, and Rui L Aguiar. Knee/elbow point estimation through thresholding. In *6th IEEE International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 413–419, Barcelona, Spain, August 2018. IEEE.

[9] S. Archak, S. Dixit, R.P. Spillane, and E. Zadok. Multi-tier caching, March 2012. US Patent App. 13/159,039.

[10] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 389–403, Renton, WA, April 2018. USENIX Association.

[11] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–75. IEEE, 2015.

[12] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. RobinHood: Tail latency aware caching - dynamic reallocation from cache-rich to cache-poor. In *OSDI*, 2018.

[13] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1—7, August 2011.

[14] Xavier Blasco, Juan M Herrero, Javier Sanchis, and Manuel Martínez. A new graphical visualization of n-dimensional pareto front for decision-making in multiobjective optimization. *Information Sciences*, 178(20):3908–3924, 2008.

[15] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. mPart: Miss-ratio curve guided partitioning in key-value stores. In *ISMM*, 2018.

[16] Yongtao Cao, Byran J. Smucker, and T. Robinson. On using the hypervolume indicator to compare pareto fronts: Applications to multi-criteria optimal experimental design. *Journal of Statistical Planning and Inference*, 160:60–74, 2015.

[17] Zhen Cao, Geoff Kuenning, and Erez Zadok. Carver: Finding important parameters for storage system tuning. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2020. USENIX Association.

[18] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, July 2018. USENIX Association. Data set at http://download.filesystems.org/auto-tune/ATC-2018-auto-tune-data.sql.gz.

[19] Zhiguang Chen, Nong Xiao, and Fang Liu. SAC: Rethinking the cache replacement policy for SSD-based storage systems. In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR)*, New York, NY, USA, 2012. Association for Computing Machinery.

[20] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, July 2015. USENIX Association.

[21] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, March 2016. USENIX Association.

[22] Dinero IV trace-driven uniprocessor cache simulator. http://pages.cs.wisc.edu/~markhill/DineroIV/.

[23] Elastic search. https://www.elastic.co/elastic-stack.

[24] Tyler Estro, Pranav Bhandari, Avani Wildani, and Erez Zadok. Desperately seeking ... optimal multi-tier cache configurations. In *HotStorage '20: Proceedings of the 12th USENIX Workshop on Hot Topics in Storage*, Boston, MA, July 2020. USENIX.

[25] Jianyu Fu, Dulcardo Arteaga, and Ming Zhao. Locality-driven MRC construction and cache allocation. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '18, pages 19–20, New York, NY, USA, 2018. ACM.

[26] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2011. USENIX Association.

[27] Alireza Haghdoost. Sim-ideal, Dec 2013. https://github.com/arh/sim-ideal/tree/master.

[28] Lulu He, Zhibin Yu, and Hai Jin. FractalMRC: Online cache miss rate curve prediction on commodity systems. *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 1341–1351, 2012.

[29] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 351–364, Denver, CO, June 2016. USENIX Association.

[30] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. Fast miss ratio curve modeling for storage cache. *TOS*, 14:12:1–12:34, 2018.

[31] Dr. Shaily Jain and Nitin Nitin. Memory map: A multiprocessor cache simulator. *Journal of Electrical and Computer Engineering*, 2012, 09 2012.

[32] Ricardo Koller, Ali José Mashtizadeh, and Raju Rangaswami. Centaur: Host-side SSD caching for storage performance control. In *IEEE International Conference on Autonomic Computing*, pages 51–60, 2015.

[33] Eugene F Krause. *Taxicab geometry: An adventure in non-Euclidean geometry*. Courier Corporation, 1986.

[34] Marco Laumanns, Lothar Thiele, and Eckart Zitzler. An efficient, adaptive parameter variation scheme for metaheuristics based on the epsilon-constraint method. *European Journal of Operational Research*, 169(3):932–942, 2006.

[35] Xiuhua Li, Xiaofei Wang, Keqiu Li, Zhu Han, and Victor CM Leung. Collaborative multi-tier caching in heterogeneous networks: Modeling, analysis, and design. *IEEE Transactions on Wireless Communications*, 16(10):6926–6939, 2017.

[36] Zhang Liu, Hee Won Lee, Yu Xiang, Dirk Grunwald, and Sangtae Ha. eMRC: Efficient miss rate approximation for multi-tier caching. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, February 2021 (to appear).

[37] Stuart Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[38] Rano Mal and Yul Chu. A flexible multi-core functional cache simulator (FM-SIM). In *Proceedings of the Summer Simulation Multi-Conference*, SummerSim '17, San Diego, CA, USA, 2017. Society for Computer Simulation International.

[39] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[40] Louis Melville Milne-Thomson. *The calculus of finite differences*. American Mathematical Soc., 2000.

[41] MySQL AB. MySQL: The world's most popular open source database. www.mysql.org, July 2005.

[42] Anant Vithal Nori, Jayesh Gaur, Siddarth Rai, Sreenivas Subramoney, and Hong Wang. Criticality aware tiered cache hierarchy: A fundamental relook at multi-level cache hierarchies. In *45th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 96–109, June 2018.

[43] Massachusetts Institute of Technology. DynamoRIO: Dynamic instrumentation tool platform, February 2009. http://www.dynamorio.org/.

[44] Sundaresan Rajasekaran, Shaohua Duan, Wei Zhang, and Timothy Wood. Multi-cache: Dynamic, efficient partitioning for multi-tier caches in consolidated VM environments. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 182–191, April 2016.

[45] T. W. Ridler and S. Calvard. Picture thresholding using an iterative selection method. *IEEE Transactions on Systems, Man, and Cybernetics*, 8(8):630–632, August 1978.

[46] Stan Salvador and Philip Chan. Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms. In *16th IEEE International Conference on Tools With Artificial Intelligence*, pages 576–584. IEEE, 2004.

[47] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a "kneedle" in a haystack: Detecting knee points in system behavior. In *31st International Conference on Distributed Computing Systems Workshops*, pages 166–171, Minneapolis, MN, June 2011. IEEE.

[48] R. P. Spillane, P. J. Shetty, E. Zadok, S. Archak, and S. Dixit. An efficient multi-tier tablet server storage architecture. In *2nd ACM Symposium on Cloud Computing*. ACM, Oct 2011.

[49] R. P. Spillane, P. J. Shetty, E. Zadok, S. Archak, and S. Dixit. An efficient multi-tier tablet server storage architecture. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11)*, Cascais, Portugal, October 2011.

[50] Shekhar Srikantaiah, Emre Kultursay, Tao Zhang, Mahmut Kandemir, Mary Jane Irwin, and Yuan Xie. MorphCache: A reconfigurable adaptive multi-level cache hierarchy. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 231–242. IEEE, 2011.

[51] Storagereview. https://www.storagereview.com/.

[52] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. *ACM Sigplan Notices*, 44(3):121–132, 2009.

[53] Elvira Teran, Zhe Wang, and Daniel A Jiménez. Perceptron learning for reuse prediction. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[54] Tea Tušar and Bogdan Filipič. Visualization of pareto front approximations in evolutionary multiobjective optimization: A critical review and the prosection method. *IEEE Transactions on Evolutionary Computation*, 19(2):225–245, 2014.

[55] David A Van Veldhuizen. Multiobjective evolutionary algorithms: Classifications, analyses, and new innovations. Technical report, AIR FORCE INST OF TECH WRIGHT-PATTERSONAFB OH SCHOOL OF ENGINEERING, 1999.

[56] Carl A. Waldspurger, Nohhyun Park, Alex Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *FAST*, 2015.

[57] Carl A. Waldspurger, Trausti Saemundson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 487–498, Berkeley, CA, USA, 2017. USENIX Association.

[58] Han Wan, Xiaopeng Gao, Xiang Long, and Zhiqiang Wang. *GCSim: A GPU-Based Trace-Driven Simulator for Multi-level Cache*, pages 177–190. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[59] John Wilkes. The Pantheon storage-system simulator, 1996.

[60] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks. In *OSDI*, 2014.

[61] Suzhen Wu, Yanping Lin, Bo Mao, and Hong Jiang. GCaR: Garbage collection aware cache management with improved performance for flash-based SSDs. In *Proceedings of the International Conference on Supercomputing*, New York, NY, USA, 2016. Association for Computing Machinery.

[62] Juncheng Yang. PyMimircache. https://github.com/1a1a11a/PyMimircache. Retrieved April 17, 2019.

[63] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI 2020)*, Virtual, November 2020. USENIX Association.

[64] Zhengyu Yang, Morteza Hoseinzadeh, Allen Andrews, Clay Mayers, David Evans, Rory Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson. AutoTiering: Automatic data placement manager in multi-tier all-flash datacenter. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, December 2017.

[65] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA: An online-model based cache allocation scheme in cloud block storage systems. In *2020 USENIX Annual Technical Conference (ATC '20)*, pages 785–798, 2020.

[66] Qinpei Zhao, Ville Hautamaki, and Pasi Fränti. Knee point detection in BIC for detecting the number of clusters. In *International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 664–673. Springer, 2008.

[67] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. da Fonseca. Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132, 2003.