Topological Analysis and Visualization of Cyclical Behavior in Memory Reference Traces

A.N.M. Imroz Choudhury*   Bei Wang†   Paul Rosen‡   Valerio Pascucci§

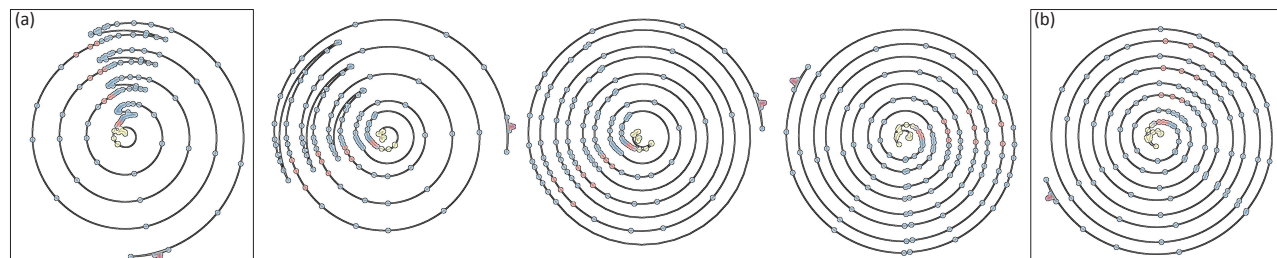Scientific Computing and Imaging Institute, University of Utah

Figure 1: One circular parameterization of the memory behavior of bubble sort (box (a)) morphing into another (box (b)), highlighting both their similarities and differences, and giving two views of the recurrent nature of the program.

ABSTRACT

We demonstrate the application of topological analysis techniques to the rather unexpected domain of software visualization. We collect a memory reference trace from a running program, recasting the linear flow of trace records as a high-dimensional point cloud in a metric space. We use topological persistence to automatically detect significant circular structures in the point cloud, which represent recurrent or cyclical runtime program behaviors. We visualize such recurrences using radial plots to display their time evolution, offering multi-scale visual insights, and detecting potential candidates for memory performance optimization. We then present several case studies to demonstrate some key insights obtained using our techniques.

Keywords: Memory reference traces, circular coordinates, visualization, topological analysis.

1 INTRODUCTION

As the gap between processor and memory speed widens [29], it becomes crucial to investigate the memory behavior of a program for better structural understanding and higher performance. Though hardware caches are used to manage this speed difference, the operation of such caches, and the memory subsystem in general, remain hidden in abstraction from the programmer, who can reason only vaguely without explicit feedback from the machine about the program's memory interactions.

If the goal is to achieve the highest possible performance through software design, it is possible to lift this curtain of abstraction, by rigging a running program to produce a *memory reference trace*, a temporally linear record of all addresses accessed by the program during its execution, indexed by logical time. Each reference represents one load or store instruction. Such a record represents the full interaction of a program with the memory, and provides a starting point for any memory analysis performed by the programmer.

However, such a raw memory reference trace is linear, whereas an application generally has non-linear programming constructs such

---

*email: roni@cs.utah.edu
†email: beiwang@sci.utah.edu
‡email: prosen@sci.utah.edu
§email: pascucci@sci.utah.edu

as branches and loops, which are evident in its source code. It is important to understand the dynamic memory behavior of a program with respect to its static source code.

In particular, a program exhibits *spatial* and *temporal locality*, the tendency to reference nearby memory locations in relatively quick succession. For example, a program exhibits *recurrent* memory access patterns, which may or may not be directly reflected through its programming constructs. For instance, a loop may execute several times, each time accessing memory locations in similar patterns, therefore inducing a circular memory access pattern. Conversely, two different stretches of the source code from two different functions may share similar memory access patterns, if they both perform similar sets of reads and writes upon the memory.

Detecting and visualizing memory access patterns can lead to a better understanding of the program behavior, as well as insights into its performance characteristics. We accomplish this goal by encoding a non-linear high dimensional structure over the memory reference trace, specifically revealing its inherent circular behavior through topological analysis, while providing a correspondence between the runtime memory behavior and the source code. Topological analysis reveals complex, multiscale features in reference traces that would be difficult to find using simple pattern matching approaches. In this paper, we propose an approach to study certain aspects of the temporal behavior of memory reference traces through topological analysis and visualization. The results in this paper are:

- We recast sequences of consecutive memory accesses within the raw memory reference trace as points in high dimensional space, therefore creating a point cloud abstraction of the temporal information encoded within the linear trace.
- We equip the point cloud with an architecturally meaningful metric, which reflects the similarity between two sequences of memory accesses, thus capturing the notion of spatiotemporal locality.
- We perform *automatic* topological analysis on the point cloud to detect circular structures which represent the recurrent, cyclical memory behaviors.
- We use *topological persistence* to guide our selection of meaningful circular structures: those with high persistence likely represent significant features within the runtime behavior of a program.
- We provide a visualization approach that connects the runtime memory behaviors with program source code. It correlates the source code with the non-linear memory behavior structures, providing insights to potential performance optimizations.

2 RELATED WORK

Even simple applications produce very large references traces which limits how they can be analyzed. Computational approaches include cache simulation [26], in which the full trace is used to produce cache statistics such as hit rates, essentially summarizing the trace by a much smaller set of measurements. Along these same lines, traces can be manipulated in such a way as to reduce their size without affecting the results they produce in a cache simulator; such approaches give either exact [1, 18] or approximate [8, 15, 23] results. These approaches work by eliminating trace records that do not substantially affect the simulation output, which is also their biggest drawback, as important, detailed structural information is lost as well, preventing the investigation of detailed patterns and structures in these traces. A different approach is to restructure traces to yield new insights. Reference affinity [31] is one such example that places a hierarchical ordering on a reference trace, grouping together correlated references as much as possible, generalizing the notion of spatial and temporal locality. The structure can be used as a guide to select caching strategies, or more generally to understand the access pattern present in a trace.

Visualization is also a common approach to understanding the content of reference traces, though most visualization approaches deal more closely with the cache than the trace itself. The Cache Visualization Tool [27] shows cache block residency, visualizing cache line contention due to the layout and access patterns of several active data structures. Yu et al. [30] use cache simulation to produce a static view of cache behavior over time. Each pixel in an image corresponds to the cache effect (hit or miss) of a single reference; as a whole, the image serves as a time-indexed "map" of cache performance. YACO [22] is a cache optimization tool focusing on performance statistics, plotting cache misses in different ways, highlighting performance bottlenecks in lines of code and data structures. By contrast, the Memory Trace Visualizer [6] visualizes reference traces directly, highlighting access patterns and showing their effects in a simulated cache, while follow-on work [7] visualizes the detailed data motion between the levels of a simulated cache.

However, all of these approaches are, in a sense, local: though they handle the flow of time via animation, at any particular moment they only demonstrate what is happening in a trace in a limited temporal range. Our goal in the current work is to find and visualize higher-order structures in reference traces that may extend through time, forming cycles that may be executed multiple times. de Silva et al. [10] and Wang et al. [28] present approaches to finding topological features, such as circles and branches, in general point sets. In the current work we adapt these approaches to a reformulation of a reference trace as a point cloud.

3 TECHNICAL OVERVIEW

Here we give a brief overview of the pipeline of data transformation that leads to our visualization. To begin, we collect a *memory reference trace*, a complete list of the memory accesses performed by a running program. We accomplish this stage using Pin [20], a binary rewriting framework that allows intercepting load and store instructions, and logging their target addresses to disk. The trace also contains information correlating individual memory accesses to line numbers in source code, allowing the visualization phase to similarly correlate memory references to source code.

The reference trace is a single-dimensional, time-indexed signal indicating the memory behavior of the program. To detect cyclical behavior, we convert the trace to a high-dimensional point cloud by considering *windows* of the trace. For some window size w, we take a sequence of w adjacent reference trace records and consider them as a point in a w-dimensional space. By collecting windows beginning at every trace record, a high-dimensional point cloud is formed. We next use topological analysis [11, 28] to detect circular features within the point cloud. These circular features represent cyclical behavior in the trace, as the points within such circles represent roughly the same pattern of memory access. This procedure produces several *parameterizations* for the point cloud, each of which indicates a different set of circular features.

Finally, we perform visualization on the various parameterizations. The parameterizations of the point cloud are circle-valued functions, allowing them to be plotted in two dimensions. Each point also has an associated logical time that is encoded in the radial direction. Finally, the source code correlation information is used to color code the plotted points, allowing a visual correlation to different parts of the program. In this view, the circular features are clearly visible while radial color correspondences indicate recurrence of particular lines of code, forming easily visible patterns. The different parameterizations visualize the same data in different ways, therefore we also provide a morphing mechanism to smoothly transition between parameterizations. Such animation is helpful for pinning down which features are compressed or expanded between different parameterizations and helps lead to a better understanding of the results. In the following sections we provide a deeper description of the analysis leading to parameterizations, their subsequent visualization, and finally demonstrate several results of our method.

4 TOPOLOGICAL ANALYSIS OF REFERENCE TRACES

Given a memory reference trace $T = (P, E)$ that combines a trace of memory operations P and the program executable E, we first encode the temporal information in P as a high-dimensional point cloud X. We then perform topological analysis on X that detects its circular features.

4.1 Encode Memory Operations as a Point Cloud

Given a set of m memory operations $P = \{p_1, p_2, ..., p_m\}$ and a window size w, we encode its temporal behavior as a high dimensional point cloud X with a metric μ as follows. We move a scanning window along P and encode every w consecutive operations as a point in w dimensions. w is the size of a scanning window that *looks ahead* w operations in time. Thus, X is a collection of n points of dimension w, $X = \{x_1, x_2, ..., x_n\}$, where $n = m - w + 1$. For each $x_i \in X$ ($1 \leq i \leq n$), $x_i = (p_i, ..., p_{i+w-1})$. Most atomic actions taken by a program result in only a few memory accesses, suggesting a window size of around 3. To capture temporal patterns, however, we expand the window size by a small factor so that each window touches multiple consecutive actions. Therefore, in our studies we have fixed $w = 10$. Though the optimal window size is probably application or trace dependent, we have gotten excellent results with this value, and we note here that the choice of window size deserves further study.

Now that we have constructed a high dimensional point cloud $X \subset \mathbb{R}^w$, we need to choose a proper distance metric μ on X. Since we care about temporal behavior of a memory reference trace, the number of *modifications* needed between two temporal windows is more important that the actual operations within each window. Therefore, given two points $x_i, x_j \in X$, the distance $\mu(x_i, x_j)$ between them is their *Levenshtein distance*. More precisely, we treat each point x_i as a string of w characters, with one character for each dimension. The *Levenshtein distance* between two ordered tuples is the minimum number of edits needed to transform one tuple into the other, where the allowable edit operations are insertion, deletion or substitution of a single element. By definition, $0 \leq \mu(x_i, x_j) \leq w$.

4.2 Detecting Circular Features in a Point Cloud

Given a high dimensional point cloud $X \subset \mathbb{R}^w$, we would like to detect its circular features. With the technical tools described in [10, 28], we now give an overview of our algorithm. Here we assume basic knowledge in topology and homology. For non-specialists, we give brief descriptions for relevant topological concepts along the way. See [21] or [16] for an easier to read background or [13] for a more computationally oriented treatment.

Suppose we represent our point cloud data X with a simplicial complex K that contains vertices, edges and triangles. Homology deals with topological features such as "cycles" in a topological space, 0-, 1- and 2-dimensional homology groups correspond to components, tunnels and voids. In a nutshell, 1-dimensional homology classes are *non-bounding cycles* represented by a collection of edges in K. *Dual* to homology groups, 1-dimensional cohomology
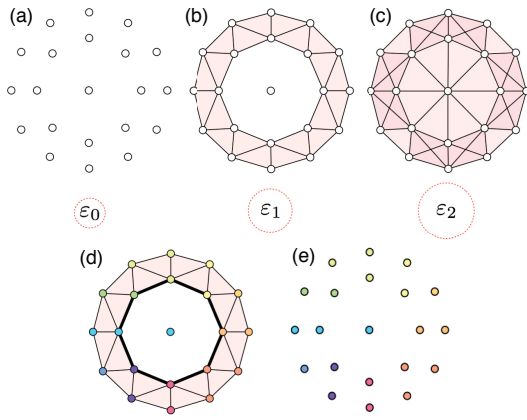
Figure 2: Algorithm pipeline: (a)-(c) data points to nested family of simplicial complexes; (d) detection of significant cohomology class and its transformation into a circle-valued function; (e) color map encoding.

classes are *non-bounding cocycles*, which are functions that map a collection of edges in K to integers.

Technically, we rely on the following principle from homotopy theory, which shows that in an algebraic way, 1-dimensional co-homology represents circular structures in data. Let $[X, \mathbb{S}^1]$ be the set of equivalence classes of continuous maps from the space X to the unit circle $\mathbb{S}^1$. Let $\mathsf{H}^1(X; \mathbb{Z})$ be the group of 1-dimensional cohomology classes with integer coefficients. For topological spaces with the homotopy type of a cell complex, there is an isomorphism (i.e. identical structure), $\mathsf{H}^1(X; \mathbb{Z}) \cong [X, \mathbb{S}^1]$ [16]. This relates cohomology with *circular coordinates*. It implies that if X has non-trivial 1-dimensional cohomology class $\alpha \in \mathsf{H}^1(X; \mathbb{Z})$, we can construct a continuous function $\theta : X \to \mathbb{S}^1$ from α (see [10] for a formal proof).

Given a point cloud $X \subset \mathbb{R}^w$, we output global circular coordinate functions $\theta : X \to \mathbb{S}^1$ that give the values for each point x in X. Our overall pipeline is as follows:

1. Represent the point cloud data X as a family of simplicial complexes.
2. Use the concept of persistent cohomology [10, 28] to detect a significant cohomology class in K, and convert such a class into a circle-valued function $\theta : X \to \mathbb{S}^1$.
3. Encode each circular coordinate in θ with a color map transfer function to highlight the circular structures.

Now we give a high-level description of each step in the above process. For the technical details, see [10, 28]. For non-specialists, we give an example afterwards as illustrated in Figure 2.

Data Points to Simplicial Complex. Point cloud data $X \subset \mathbb{R}^w$ with a metric μ can be represented as a single simplicial complex, or more usefully as a nested family of simplicial complexes [9]. We use the Vietoris-Rips complex, $\mathrm{Rips}(X, \varepsilon)$, where there is a p-simplex for every finite set of $p+1$ points in X with diameter at most ε. Since we are only interested in computing H^1, we use its 2-skeleton, that is, the vertices, edges and triangles. For $\varepsilon_0 \le \varepsilon_1 \le ... \le \varepsilon_n$, we obtain a nested family of simplicial complexes, $\mathscr{K} : K(\varepsilon_0) \subseteq ... \subseteq K(\varepsilon_n)$, where $K(\varepsilon_i) = \mathrm{Rips}(X, \varepsilon_i)$.

Simplicial Complex to Circular Coordinate Function. Now we are given a nested family of simplicial complexes that represent the structure at different parameter values ε. We introduce the notion of *spatial scale* for learning the structure through the concept of *persistsence*. Persistence studies the evolution of vectors in a sequence of vector spaces [4]. One main example of such a sequence comes from the cohomology groups of a nested sequence of simplicial complexes constructed at different scales. Persistence provides a way of ranking the significance of the cohomology classes and is essential to achieving robustness of the proposed methods. Intuitively, persistence separates features from noise by measuring the
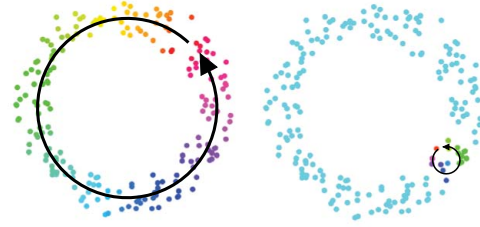

Figure 3: The circular structure on the left has high persistence while the one on the right is considered topological noise [10].

significance (e.g. size) of circular structures. An illustrative example is shown in Figure 3, where the feature on the left corresponds to high persistence, or significant circle structure, while the feature on the right might be considered topological noise.

The algorithm that computes the persistent cohomology of a sequence of simplicial complexes [11] is a modified version of the persistent homology algorithm [3, 14], which in turn is a variation of the classic Smith normal form algorithm [21]. It involves a specific ordering in conducting matrix reduction on the coboundary matrices of the nested simplicial complexes. After the matrix reduction, we obtain a collection of cocyles, each represented as a set of edges with coefficients. Each cocycle is then transformed into a circle-valued coordinate function $\theta : X \to \mathbb{S}^1$ through lifting, smoothing and integration using well-established procedures [10]. For the story behind persistent homology, see [12, 13].

Color Map Encoding. Each circular coordinate function $\theta : X \to \mathbb{S}^1$ is then encoded with a color map transfer function to highlight the corresponding circular structure (Figures 3 and 4). For a high-dimensional point cloud X, a dimension reduction technique such as ISOMAP [25] is applied first, in order to project X onto a low-dimensional space of dimension 2 or 3. However, as shown in Figure 5(a), color map encoding serves as a naive visualization of the circular structures in the point cloud data. For better circular structure visualization and visual analytics, we apply the techniques discussed in Section 5.

Example. To illustrate persistence and our pipeline, we give an example shown in Figure 2. In Figure 2(a)-(c), for $\varepsilon_0 < \varepsilon_1 < \varepsilon_2$, we build a nested family of simplicial complexes, $\mathscr{K} : K(\varepsilon_0) \subseteq K(\varepsilon_1) \subseteq K(\varepsilon_2)$. For a small diameter ε_0 in (a), no vertices are connected in the Vietoris-Rips complex, therefore $K(\varepsilon_0)$ contains only vertices from the original point cloud. For a slightly larger diameter ε_1 in (b), some edges and triangles appear in $K(\varepsilon_1)$, giving *birth* to a non-trivial circular structure (represented by a 1-dimensional cocycle) that looks like a tunnel within an annulus. For a larger diameter ε_2 in (c), the circular feature in the middle of the space gets filled in and *dies* (disappears). The *persistence* of such a feature is its death time minus its birth time, that is, $\varepsilon_2 - \varepsilon_1$. If ε_2 is much larger than ε_1, we consider the above circular feature (cocycle) to be significant. We then fix our attention on the simplicial complex when such a feature appears, that is, $K(\varepsilon_1)$, where such a cocycle is then computed and transformed to a circle-valued coordinate function, as shown in
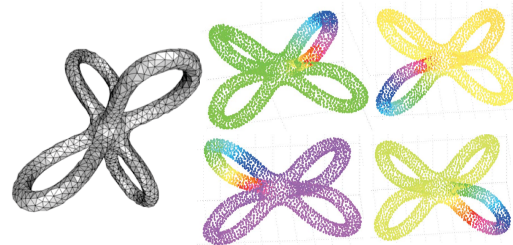

Figure 4: A point cloud X is sampled from a genus-4 surface. We construct four circle-valued coordinate functions that correspond to its significant circular structures, visualized by color map transfer functions.
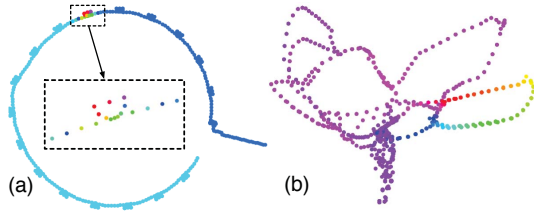
Figure 5: Two reference traces analyzed using our topological methods and projected onto 2D using ISOMAP. Their circular features are visualized by color map. (a) A small trace showing 18 different circular structures, with one of them visualized by color map. (b) The naive ISOMAP embedding of a circular structure that is better represented using our visualization methods in Figure 10(a).
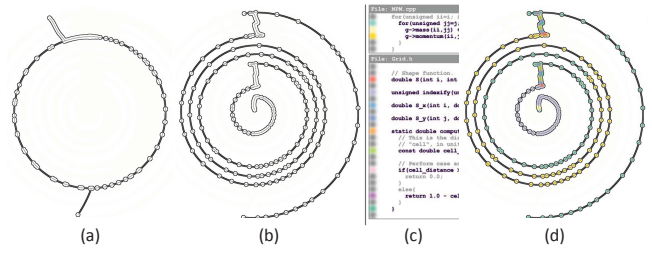

Figure 6: Available visualization options for parameterized data. (a) The parameterization is placed in a circular structure. (b) Time is applied to the radius. (c-d): The data points are correlated back to source code using a color map.

Figure 2(d). Furthermore, it is encoded with a color map transfer function, as shown in Figure 2(e).

Parameter Selection and Limitations. We require only one parameter, ε, in our topological analysis. It is used in computing the 2-skeleton of the Vietoris-Rips complex. In particular, for $\varepsilon = \infty$, computing the 2-skeleton of the Vietoris-Rips complex for n high-dimensional points results in $O(n^3)$ simplices, giving a worst-case time complexity of $O(n^3)$. This is, however rare in practice [32]. The persistence algorithm runs in time $O(v^3)$, where v is the number of simplices [5]. However this takes roughly linear time in practice [2]. For a detailed performance analysis, see [28]. Usually ε is chosen with prior knowledge of the problem domain, to be just large enough to detect the topology. This decreases the above bound to an expected linear or even constant behavior.

5 VISUALIZING CYCLES IN REFERENCE TRACES

As stated, dimension reduction techniques can work well for simple circular structures. The technique can fail however when the high-dimensional structure of the memory reference trace becomes complicated. Figure 5(b) shows an example where dimension reduction produces a visualization which is extremely difficult to interpret. What is needed instead is a visualization approach that is more robust to complex structure.

5.1 Circular Visualization

As an alternative, we propose using the circular parameterization θ of all points as means of formulating a visualization. The parameter θ is first conditioned by performing a scale and offset such that $\theta \in [0 : 2\pi]$. θ contains points which are cycle members, but may also contain non-members in the form of a preamble and post-amble which need to be separated from the cycle. This is accomplished by selecting the first and last members of θ as θ_{pre} and θ_{post} respectively. Neighboring parameterizations are collected into the pre- and post-amble while $|\theta_i - \theta_{pre}| < \varepsilon$ or $|\theta_i - \theta_{post}| < \varepsilon$, respectively. Finally, points are mapped to the image by placing cycle members on a fixed radius circle, while the pre- and post-ambles then have their radii set to increase monotonically away from the center of the output domain.

Finally, the visualization is assembled. Points which are temporal neighbors are connected using arcs that undergo polar interpolation. Isocontouring is then applied to form a summary structure. Figure 6(a) shows an example of this visualization which indicates that we do truly have a circular structure.

5.2 Spiral Visualization

Unfortunately, while the circular visualization summarizes the parameterization, it fails to illuminate many interesting structures within it. We have extended this visualization to be more descriptive by adding an additional dimension of activity to the visualization.

From the visualization standpoint, the visual information channel represented by the radius r was underutilized while from the parameterization standpoint, the temporal relationship between points was underutilized as well. A more informative visualization is created by linking the temporal property of the parameterization with the radius of the output point by simply varying the radius r_i of θ_i by the value i. The radius then encodes time, with earlier events appearing in the center of the output domain and later events appearing towards the periphery. Figure 6(b) demonstrates this capability, producing a far more informative visualization than the circular visualization from the prior section.

Correlation with Source Code. While the circular structures discovered by our method are interesting, it is difficult to interpret their meaning as a standalone representation. As a final addition to our visualization, we provide users a color coding system that correlates the structures that have been discovered with the familiar context of source code. Figure 6(c)-(d) show the result of this final stage. Once the color coding is in place it becomes more obvious what programmatic structures correlate to the circular structures. We also offer the ability to collapse program structures (e.g. functions) such that groups of source code elements can also be grouped by color in the visualization.

Morphing Between Parameterizations. Most reference traces produce multiple parameterizations θ^j. These various parameterizations can have many relationships to one another. They may highlight structures of different scales (outer loops versus inner loops), they may be duals of one another (pointing to some kind of interleaving operations), or they may be entirely unrelated. Morphing between two parameterizations gives the opportunity to better identify these relationships. Since the time associated with individual points does not change between parameterizations, we have $r_i^j = r_i^k$ for each point. To morph between the parameterizations, we vary the angle of each point θ_i between θ_i^j and θ_i^k. Figure 1 shows an example. The points of the visualization are interpolated in a coherent manner, but the connecting structure may having popping effects as the geometry switches from rotating clockwise to counterclockwise or vise-versa. For additional examples, see the supplemental video.

Data Set	Original Trace Size	Records Used	Sample Interval	Persistence Rank	Total Parameterizations
Bubble sort using vector (Fig. 7)					
Sorted	141K	680	0	(a) 1 (b) 1	14
Reverse	141K	1275	0	(c) 2 (d) 1	6
Shuffled	141K	1115	0	(e) 2 (f) 1	2
Bubble sort using array (Fig. 7)					
Sorted	141K	460	0	(g) 1	6
Reverse	141K	690	0	(h) 1	3
Matrix Multiply (Fig. 9)					
Standard	173 K	1000	0	(a) 3 (b) 2 (c) 1	4
Blocked	92 K	1000	0	(d) 2 (e) 1 (f) 1	4
Material Point Method - 5 particle (Fig. 10)					
Interpolation	1.0 M	2000	0	(a) 1 (b) 1	9
Material Point Method - 60 particle (Fig. 11)					
Fig. 11 (a)	2.8 M	1000	0	(a) 2	2
Fig 11 (b)-(c)	2.8 M	1000	10	(b) 3 (c) 1	5

Table 1: Details of the data used in our experiments.

6 RESULTS

We now illustrate our results on several memory reference traces using the proposed methods, focusing on different kinds of program structures. The first data set performs bubble sort on a list of numbers. The second uses the first half of a trace originating from a program that performs matrix multiplication. The third data set comes from a material point method (MPM) simulation code, which involves particles moving on a grid.

Table 1 enumerates the details of the data used for our experiments. The processing of data through our system takes on the order of a few minutes, at most. The most time consuming component is collecting the memory trace, which takes on the order of a few seconds to a few minutes for our examples. Computing the parameterizations takes on the order of seconds, and the visualization renders at highly interactive rates. We use topological persistence to guide our selection of meaningful circular structures. In general, all circular features selected have high persistence rankings, indicating their significance.

In the following sections, we demonstrate that our method offers insights into various non-linear memory behavior structures by connecting the source code with topological analysis and visualization.

6.1 Analyzing Loop Contents

We start by analyzing loop-based recurrent behavior within a bubble sort data set. Bubble sort works by repeatedly sweeping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. These sweeps become progressively shorter as items are sorted into place. For example, as shown in Figure 7(left column), sorting through a list of five ascending-ordered numbers results in 4 standalone comparisons, while sorting through a list of five descending-ordered numbers performs 10 comparisons followed by swaps. Sorting the given list of randomly shuffled numbers results in 6 comparisons followed by swaps, and 4 standalone comparisons.

Figure 7 shows various recurrent runtime structures captured by the proposed method, with each image highlighting some specific features of the algorithm computation.

Figures 7(a) and 7(b) represent memory structures within sorting five ascending-ordered numbers. Figure 7(a) shows 4 circular structures corresponding to the comparison operation occurring in the *if* statement (line 5). The circle itself represents one of the two C++ Standard Template Library (STL) vector lookups, while the zig-zag secondary feature corresponds to the other. It serves to distinguish the 2 instances of vector lookups while keeping the recurrent nature of comparisons in view. On the other hand, Figure 7(b) shows each lookup on its own circle. There are 2 vector lookups per comparison, with 4 independent comparisons leading to a total of 8 circles. This image shows that the 2 vector lookups occurring per comparison have almost identical memory signatures, which is not directly evident in the source code. To better understand the correlations between these two circular features, we construct a morphing between them as shown in Figure 1, which showcases the expansions of secondary features from Figure 7(a) to (b).

Figure 7(c) and 7(d) show a bubble sort proceeding on a descending-ordered list. In (c), the outer loop (line 2) forms the 4 circles, while the 10 comparisons followed by swaps appear as teeth-like features within each repetition, showcasing the properties of the input data, and the corresponding computational structure of the algorithm. By contrast, (d) indicates the recurrent structure of the inner loop (line 4). Each of the 10 bundled circular structures contains 3 circular sub-structures, which reflect 2 vector lookups in the comparison (line 5) and 1 swap (line 6). Here, the appearance of properly spaced bundles serves to separate the important recurring features. Figure 7(e) represents the sorting of five random-ordered numbers. The circular structures correspond to the 6 comparisons followed by swaps, while the blue teeth-like features indicate the 4 standalone comparisons that are *not* followed by swaps. The analysis is able to pick up on this feature, encoded in the neighborhood information in the point cloud, which turns out to have significance in this program. By contrast, Figure 7(f) is the dual of Figure 7(e),
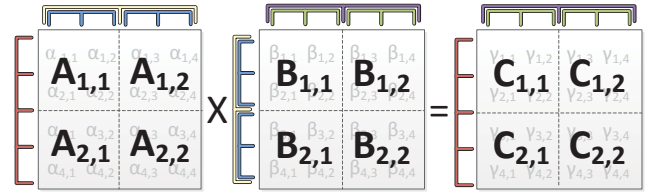

Figure 8: The block and loop structure of blocked matrix multiplication.

showing the non-swapping comparisons as cycles, and the swapping comparisons as teeth-like features.

For comparison, Figure 7(g-h) shows a version of the bubble sort program that uses bare arrays rather than STL vectors. Note that Figures 7(g) and (h) have similar memory signatures as Figures 7(b) and (c), respectively, but with fewer memory accesses. Although the volume of memory access changes due to modifications of data structures, the characteristics of the memory behavior stay the same.

6.2 A Closer Look at Nested Loops

For the bubble sort described above, the main work loop is repeated a fixed number of times, while variations in input and computation are shown as various features within the visualization. We are also interested in more complex loop structures such as those nested loops found in the matrix multiplication. As shown in Figure 9, there are two types of methods developed for matrix multiplication: the standard (top source code) and the blocked implementation (bottom source code). The blocked implementation operates on small blocks (or sub-matrices) of data that can fit into cache and be used repeatedly. For example, as shown in Figure 8, two 4-by-4 matrices A and B are multiplied with a block size of 2, and the blocked implementation operates on sub-matrices of A and B, accumulating the results in the matrix product C. Given an matrix A and B, each with 2 row partitions and 2 column partitions, their product C with 2 row partitions and 2 column partitions can be calculated by $C_{ij} = \sum_{k=1}^{2} A_{ik} B_{kj}$, where A_{ij}, B_{ij} and C_{ij} ($1 \le i, j \le 2$) are their corresponding partitions. Our implementation is a slight variation on this basic algorithm, using only 5 nested loops instead of 6, as it appears in Hennessy and Patterson [17].

Since the standard implementation uses triply nested loops, Figures 9(a-c) show its various runtime structures at three different scales. Circular structures in (a) correspond to the outermost i loops (*matmult.cpp*, line 2), while the intermediate j loops (*matmult.cpp*, line 3) are compressed into teeth-like features that oscillate as they move out radially, and the innermost k loops (*matmult.cpp*, line 4) are compressed into linear features. The j loops dominate as circular features in (b), encoding the compressed k loops as teeth-like features and the i loops as a single green point. Finally in (c), the k loops along with the multiplication itself in the innermost loop (*matmult.cpp*, line 5) becomes visible. In particular, the bundled circular structures are well-spaced, showing 4 matrix accesses for each iteration of the k loop. The shifting in alignment between the 4th and the 5th bundles indicates the slight change in memory locations required in moving to the next row. This example demonstrates how the same sequence of memory events can be parameterized in different scales, where as a single class of event rises to prominence, the other events are compressed as secondary features. The analysis focuses on the various loops because of their self-similarities and heavy recurrences, which provide meaningful context for the software engineer.

By comparison, Figure 9(d-f) gives a glimpse of the runtime structures of a block matrix multiplication by focusing on the 3 innermost loops. Circular structures in (d), (e) and (f) correspond to the i, k, and j loops (*blocked-matmult.cpp*, lines 4, 5 and 7), respectively, while compressing the other loops into secondary features. In particular, (f) showcases bundled circular structures, where each bundle represents the two instances of the j loop due to blocking.
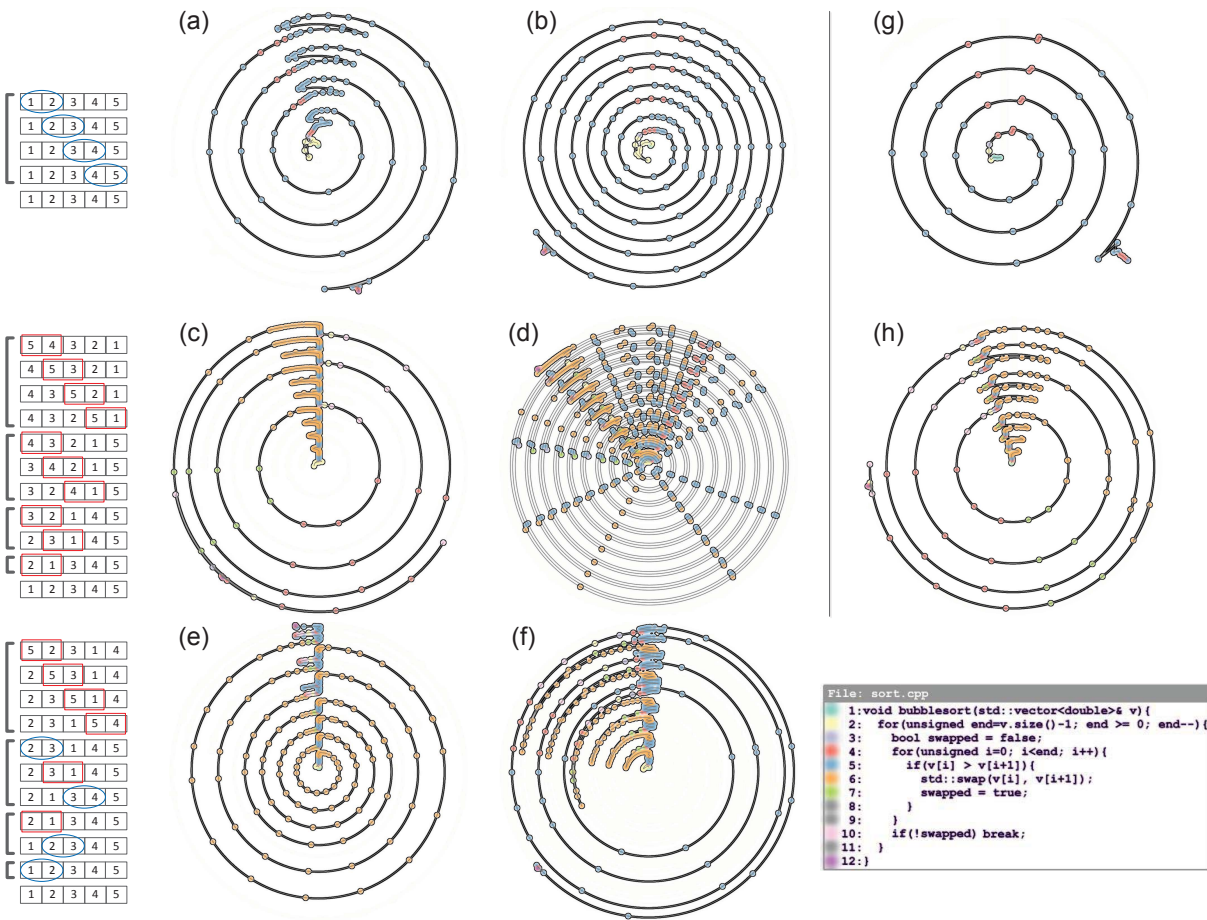
Figure 7: Various recurrent runtime structures within a bubble sort of five numbers ordered (a-b) ascendingly, (c-d) descendingly, and (e-f) randomly. (g-h) Versions of (b-c) in which a bare array has been used in place of STL vectors, eliminating many overhead memory accesses associated with the STL.

6.3 Non Loop-Based Recurrent Behavior

Now that we have seen loop-based recurrent features in the previous sections, we move on to study non loop-based recurrent behavior, e.g. that of repeated calls to the same function, or to different functions with similar or identical memory access patterns, such as those found within a material point method (MPM) simulation code. MPM [24] simulates solid bodies, modeled as collections of particles, moving in response to applied loads. The particles carry physical attributes such as mass, velocity, stress, etc., which in one phase of the algorithm, can be interpolated to a fixed background grid to compute spatial gradients, as necessary for solving the equations of motion.

Figure 10 shows the parameterized recurrent structures during the interpolation phase, where mass and momentum are being interpolated from the particles to the 2D grid nodes via an interpolation kernel, the so-called *shape function*. In Figure 10(a), the mass (*MPM.cpp*, line 3) and momentum (*MPM.cpp*, line 4) of a single particle are interpolated onto 3 grid nodes, respectively, where each of the 3 grid nodes makes 2 calls to the shape function (*MPM.cpp*, lines 3 and 4), resulting in 6 completely bundled circular features. In particular, each bundle reflects the dimensionality of the simulation. Upon close inspection, each bundle corresponds to one call to the shape function, which internally defers to one *x* directional (*Grid.h*, line 3) and one *y* directional (*Grid.h*, line 4) function call. These two functions differ only in the directional grid spacing, and therefore have extremely similar memory access patterns that are picked up by our analysis and visualization. Finally, Figure 10(b) is dual to Figure 10(a), compressing the latter's circular features to reflect the repeated calls to the *indexify* function (*Grid.h*, line 2) instead, which

calculates linear index for multidimensional data. For additional examples, see the supplemental document.

6.4 Analyzing Large Traces

Often we are interested in recurrent structure over much longer periods of time, and for such cases we use sampling techniques to extend the effective range of our methods. Figure 11(a) shows a visualization of 1000 reference trace records of a 60 particle MPM run, capturing only the first 10 initializations of the mass and momentum variables. On the other hand, we sample 10000 trace records by choosing every 10th record. Figure 11(b) shows the resulting 1000 records, spanning ten times the duration of Figure 11(a). The highly regular patterns in the 3 cycles reflect strong recurrent behavior on a larger time scale, capturing all 60 initializations of the mass and momentum variables, while showing the remainder of the memory activity as a rising linear feature. Because of this *time compression* effect, each of these cycles no longer correlates directly to a specific line of source code, but rather expresses general program structures. Figure 11(c) is the dual of (b), showing the initialization phase as a linear structure, and expanding the remainder of the trace, interpolation of the mass and momentum to the background grid, as time-sampled cycles.

By increasing the sampling interval, we are able to display a much longer trace while keeping the ability to distinguish different parts of the program. Sampling the trace allows the analysis to find large-scale structures, providing a picture of the entire run of a program, rather than details of individual functions, loops, or lines of code. As such, sampling can be used to manage level-of-detail for reference traces.

7 DISCUSSION

Performance Optimizations. When the visualization reveals recurrent runtime behavior that reflects the repetitive nature of a portion of the program, it can suggest potential performance optimizations. For example, in Figure 10(a) the two bundled circles represent nearly identical function executions, differing only in the value of a single parameter, suggesting that the two executions could coalesce into one, sparing the duplication of several computations and memory accesses. This is similar to the idea of *loop fusion* [19], in which loop bodies from independent loops may be combined to eliminate loop overheads and gain possible caching benefits from increased data reference locality. Knowing whether transformation would increase performance requires further study, but the focus of our technique lies in highlighting the possibility, which is much harder to see with existing techniques.

Our approach also reveals the circular structures of program constructs usually hidden by programming abstractions, such as helper functions, standard libraries, or operator overloading. For example, as shown in Figure 7(d), in the bubble sort case study, using the

File: matmult.cpp

```
1:    unsigned int i, j, k;
2:    for (i = 0; i < N; i++)
3:      for (j = 0; j < N; j++)
4:        for (k = 0; k < N; k++)
5:          linC[i*N + j] += linA[i*N + k] * linB[k*N + j];
```

File: blocked-matmult.cpp

```
1:    unsigned int i, j, k, j0, k0;
2:    for (k0 = 0; k0 < N; k0 += b)
3:      for (j0 = 0; j0 < N; j0 += b)
4:        for (i = 0; i < N; i++)
5:          for (k = k0; k < min(k0 + b, N); k++) {
6:            r = linA[i*N + k];
7:            for (j = j0; j < min(j0 + b, N); j++)
8:              linC[i*N + j] += r*linB[k*N + j];
9:          }
```
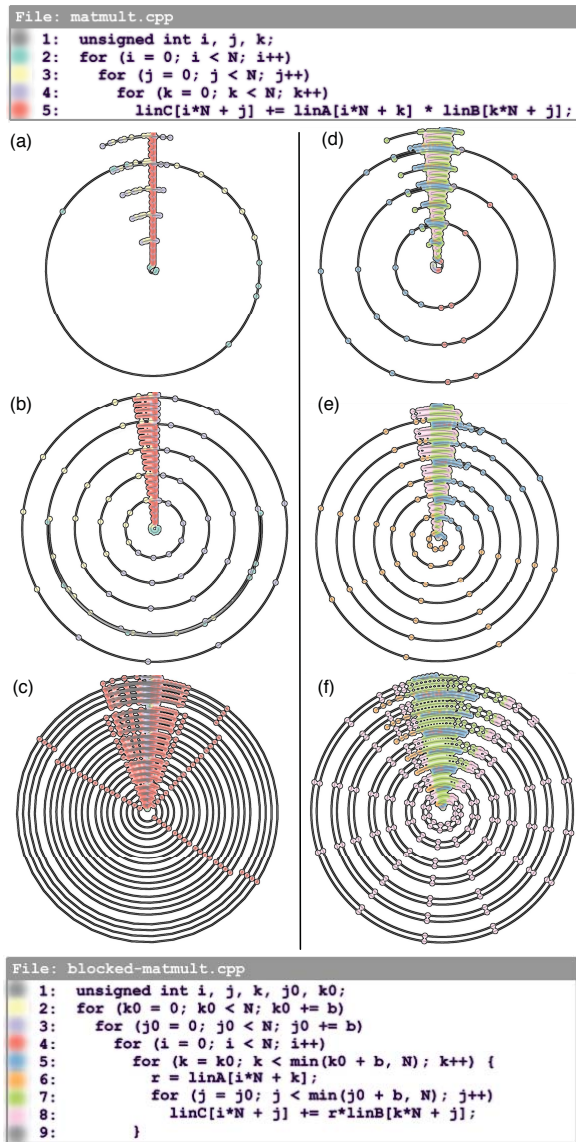
Figure 9: Various recurrent runtime structures within matrix multiplication algorithms. (a-c) Standard matrix multiplication. (d-f) Blocked matrix multiplication. Top: source code for standard implementation. Bottom: source code for blocked implementation.

File: MPM.cpp

```
1:for(unsigned ii=i; ii<=i+1; ii++){
2:  for(unsigned jj=j; jj<=j+1; jj++){
3:    g->mass(ii,jj) += g->S(ii, jj, mp->position(p))*mp->mass(p);
4:    g->momentum(ii,jj) += g->S(ii, jj, mp->position(p))*mp->mass(p)*mp->velocity(p);
5:  }
6:}
```

File: Grid.h

```
1:double S(int i, int j, const Point& p){ ... }
2:unsigned indexify(unsigned i, unsigned j) const { ... }
3:double S_x(int i, double x){ ... }
4:double S_y(int j, double y){ ... }
5:static double compute_shape_function(int cell, double position, double cell_size){
6:  // This is the distance of "position" from the position of "cell".
7:  const double cell_distance = std::abs((position - cell_size*cell) / cell_size);
8:  // Perform case analysis.
9:  if(cell_distance >= 1.0){
10:     return 0.0;
11:  }
12:  else{
13:     return 1.0 - cell_distance;
14:  }
15:}
```
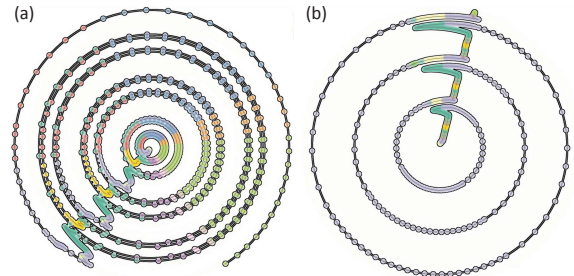

Figure 10: Interpolating mass and momentum in MPM. (a) Interpolation operation for a single particle. (b) A dual view of (a), expanding a non-interpolation action into the circular structures.

STL vectors involves many more memory accesses than the naive implementation. Prior to running our analysis, we were not fully conscious of such complexities within the STL library. Our visualization may suggest places where such extra memory references can be eliminated. Though other techniques exist for simply counting memory accesses, our approach highlights the difference visually while pointing out the cyclical nature of both programs, thus adding value over previous approaches.

In short, our tool would aid understanding of the correlations among different portions of the code, revealing hidden program constructs, and identifying potential candidates for performance optimizations. Nevertheless, further study of the potential application of this approach to performance optimization is needed.

Topological Persistence. Once the parameter ε is chosen, we perform *automatic* topological analysis to detect circular structures. Once running, the analysis does not require fine-tuning of parameters or user intervention. We then use topological persistence to guide our selection of meaningful circular structures. As our examples strongly suggest, those with high persistence likely represent significant features within the runtime behavior of a program. As displayed in Table 1, all significant circular features presented have highly ranked persistence. Furthermore, there is a clear separation in persistence measures between interesting and trivial circular features.

8 CONCLUSION

In conclusion, we have presented a general framework for exploring and discovering recurrent behavioral patterns in memory reference traces. We first recast a list of reference trace records, a staple of software memory analysis, as a high-dimensional point cloud. We then employ topological analysis to detect its circular features. The novelty of our work lies in (a) the design of a proper metric that allows the computation of *meaningful* circular structures, based on the nature of source code and the program runtime behavior, and (b) the application of topological analysis of circular features within the field of *software visualization*. We have demonstrated that both loop-based and non loop-based recurrent structures can be captured by our analysis and visualization. While the former confirms the expected structures of a program, the latter highlights less obvious features that are possible candidates for performance optimization.

We consider several extensions to our work as follows.

Visualization. From the aspect of visualization, it would be interesting to combine salient features from different parameterizations into

```
File: MPM.cpp
 1:  // Clear the mass and momentum values from the grid.
 2:  for(unsigned i=0; i<g->numNodes(); i++){
 3:      g->mass(i) = 0.0;
 4:      g->momentum(i) = Vector(0.0, 0.0);
 5:  }
 6:  // Interpolate mass and momentum of the points to the grid.
 7:  for(unsigned p=0; p<mp->numMaterialPoints(); p++){
 8:      // Compute the grid cell in which the point resides.
 9:      unsigned i, j;
10:      g->enclosingCell(mp->position(p), i, j);
11:      for(unsigned ii=i; ii<=i+1; ii++){
12:          for(unsigned jj=j; jj<=j+1; jj++){
13:              g->mass(ii,jj) += g->S(ii, jj, mp->position(p))*mp->mass(p);
14:              g->momentum(ii,jj) += g->S(ii, jj, mp->position(p))*mp->mass(p)*mp->velocity(p);
15:          }
16:      }
17:  }
```
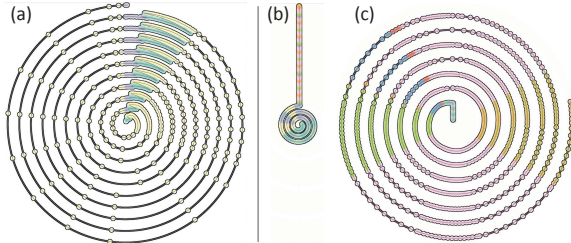

Figure 11: Analyzing the MPM trace at a larger scale. (a) 1000 consecutive trace records (no sampling). (b) 1000 trace records produced by sampling 1 of every 10 records from a segment of 10000 consecutive trace records. (c) A dual view of (b) showing recurrences later in the trace.

a single view such that the user can expand and collapse particular circular features during exploration. Selecting the interesting parameterizations to be spliced together remains a challenge. Providing additional interaction modes such as brushing and linking would be beneficial. Finally, a usability study is needed to judge both the interaction and the usefulness of our approach.

Adaptive Sampling of Memory Reference Traces. In the previous section, we have discussed a naive subsampling of the memory trace in an effort to analyze larger program structures. One possible alternative to this approach is to use source code or previous analysis results as a guide for adaptive sampling of the source code. For example, many functions have deterministic memory access patterns that can be found using either the source code or previously discovered structures. Using this knowledge, those portions of the memory trace could be removed or sampled less frequently in further analysis, thereby revealing new, larger structures in the memory trace.

REFERENCES

[1] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5:78–101, 1966.

[2] P. Bendich, H. Edelsbrunner, and M. Kerber. Computing robustness and persistence for images. *IEEE Trans. Vis. Comput. Graph.*, 16:1251–1260, 2010.

[3] G. Carlsson, A. J. Zomorodian, A. Collins, and L. J. Guibas. Persistence barcodes for shapes. *Proc. Eurographs/ACM SIGGRAPH Symposium on Geometry Processing*, pages 124–135, 2004.

[4] F. Chazal, D. Cohen-Steiner, M. Glisse, L. J. Guibas, and S. Y. Oudot. Proximity of persistence modules and their diagrams. *Proc. 25th Ann. Sympos. Comput. Geom.*, pages 237–246, 2009.

[5] C. Chen and M. Kerber. Persistent homology computation with a twist. *Proc. 27th European Workshop Comput. Geom.*, 2011.

[6] A.N.M. I. Choudhury, K. C. Potter, and S. G. Parker. Interactive visualization for memory reference traces. *Comput. Graph. Forum*, 27:815–822, 2008.

[7] A.N.M. I. Choudhury and P. Rosen. Abstract visualization of runtime memory behavior. *Proc. 6th IEEE Int. Workshop on Visualizing Software for Understanding and Analysis*, 2011.

[8] E. G. Coffman and B. Randell. Performance predictions for extended paged memories. *Acta. Inform.*, 1:1–13, 1971.

[9] V. de Silva and G. Carlsson. Topological estimation using witness complexes. *Symp. on Point-Based Graph.*, pages 157–166, 2004.

[10] V. de Silva, D. Morozov, and M. Vejdemo-Johansson. Persistent cohomology and circular coordinates. *Proc. 25th Ann. Sympos. Comput. Geom.*, pages 227–236, 2009.

[11] V. de Silva, D. Morozov, and M. Vejdemo-Johansson. Dualities in persistent (co)homology. Manuscript, 2010.

[12] H. Edelsbrunner and J. Harer. Persistent homology - a survey. *Contemp. Math.*, 453:257–282, 2008.

[13] H. Edelsbrunner and J. Harer. *Computational Topology: An Introduction*. American Mathematical Society, Providence, RI, USA, 2010.

[14] H. Edelsbrunner, D. Letscher, and A. J. Zomorodian. Topological persistence and simplification. *Discrete Comput. Geom.*, 28:511–533, 2002.

[15] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. *ACM SIGMETRICS Perform. Eval. Rev.*, 25:115–126, 1997.

[16] A. Hatcher. *Algebraic Topology*. Cambridge University Press, 2002.

[17] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd edition, 2003.

[18] S. F. Kaplan, Y. Smaragdakis, and P. R. Wilson. Flexible reference trace reduction for VM simulations. *ACM Trans. Model. Comput. Simul.*, 13:1–38, 2003.

[19] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. *Proc. 6th Int. Workshop on Lang. and Compilers for Par. Comp.*, pages 301–320, 1994.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pages 190–200, 2005.

[21] J. R. Munkres. *Elements of algebraic topology*. Addison-Wesley, Redwood City, CA, USA, 1984.

[22] B. Quaing, J. Tao, and W. Karl. Yaco: A user conducted visualization tool for supporting cache optimization. *Proc. High Perf. Comp. and Comm.*, pages 694–703, 2005.

[23] A. J. Smith. Two methods for the efficient analysis of memory address trace data. *IEEE Trans. Softw. Eng.*, 3:94–101, 1977.

[24] D. Sulsky, S.-J. Zhou, and H. L. Schreyer. Application of a particle-in-cell method to solid mechanics. *Comput. Phys. Comm.*, 87:236–252, 1995.

[25] J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

[26] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *ACM Comput. Surv.*, 29:128–170, 1997.

[27] E. van der Deijl, G. Kanbier, O. Temam, and E. D. Granston. A cache visualization tool. *Computer*, 30:71–78, 1997.

[28] B. Wang, B. Summa, V. Pascucci, and M. Vejdemo-Johansson. Branching and circular features in high dimensional data. *IEEE Trans. Vis. Comput. Graph.*, 17:1902–1911, 2011.

[29] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Comput. Arch. News*, 23:20–24, 1995.

[30] Y. Yu, K. Beyls, and E. D'Hollander. Visualizing the impact of the cache on program execution. *Information Visualization*, pages 336–341, 2001.

[31] Y. Zhong, X. Shen, and C. Ding. A hierarchical model of reference affinity. *Proc. 16th Int. Workshop on Lang. and Compilers for Par. Comp.*, 2003.

[32] A. J. Zomorodian. Fast construction of the Vietoris-Rips complex. *Comput. Graph.*, 34:263–271, 2010.