Data-driven Spatial Locality

Svetozar Miucin
University of British Columbia
Vancouver, Canada
smiucin@ece.ubc.ca

Alexandra Fedorova
University of British Columbia
Vancouver, Canada
sasha@ece.ubc.ca

ABSTRACT

Researchers and practitioners dedicate a lot of effort to improving spatial locality in their programs. Hardware caches rely on spatial locality for efficient operation; when it is absent, they waste memory bandwidth and cache space by fetching data that is never used before it is evicted. Improving spatial locality is difficult. For the most part, these are manual efforts by expert programmers, requiring substantial insight into the program's data layout and data access pattern.

This work introduces Access Graphs: a novel abstraction of memory access patterns that exposes spatial locality features and allows for automatic computation of better memory layouts. Using access graphs and a set of analysis algorithms and tools, we are able to significantly improve simulated cache miss rates by changing data layout. Further, we use random forest classifiers to automatically identify features of the data that correlate with how the data is actually used. We build a memory allocator that uses these features to guide data allocation at runtime and achieves a better spatial locality and improved performance as a result.

CCS CONCEPTS

• Theory of computation → Data structures design and analysis; **• Hardware → Memory and dense storage**; **• Software and its engineering → Software performance**; **• Computing methodologies →** *Machine learning algorithms*; Simulation evaluation;

KEYWORDS

memory performance, spatial locality, graph algorithms, random forests

ACM Reference Format:
Svetozar Miucin and Alexandra Fedorova. 2018. Data-driven Spatial Locality. In *The International Symposium on Memory Systems (MEMSYS), October 1–4, 2018, Old Town Alexandria, VA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3240302.3240417

1 INTRODUCTION

Memory wall is the phenomenon where the cost of memory accesses exceeds the cost of non-memory instructions to the point that the program spends most of its CPU time waiting on memory. Wulf and McKee proposed that modern software would hit the memory wall in the beginning of this millennium [31]. Ailamaki, DeWitt, Hill and Wood showed in 1999 that database systems spent 20-50% of CPU time waiting on memory [1]. For modern "cloud" workloads this figure is 50% on avearage, and reaches 90% for OLTP benchmarks [12]. Caches mitigate memory latency, but it is believed that they will never catch up with the voracious appetite of modern applications [12].

To navigate the limitations of hardware, programmers invest substantial effort into software optimizations aimed at reducing cache miss rates. One family of such optimizations is about rearranging the program data in memory in order to improve spatial locality. Spatial locality occurs when data items that are accessed close together in time also happen to reside close together in memory. Hardware caches fundamentally rely on spatial locality for efficient operation. Finding an optimal arrangement of objects in memory is NP-hard. A guiding principle used in prior work on memory layouts is ***to put objects that are frequently accessed together close to each other in the address space***. Literature review has revealed that these optimizations are largely manual and require deep understanding of the program's algorithms and data structures, making many of them the subjects of top-tier publications [3, 7, 10, 14, 15, 17, 21, 27, 29, 30, 33, 34]. These algorithms deliver significant performance improvements, but are very difficult to implement.

Many memory layout optimization algorithms rely on using some features of data itself to inform the placement of objects in memory. For example, it is common in mesh traversal algorithms to pack mesh nodes and triangles according to their in-domain proximity – objects with similar Cartesian coordinates. We aim to generalize this approach to any program that operates on many objects in memory and automate the extraction of knowledge needed to derive new layout strategies.

This work introduces access graphs – a novel representation of a program's memory access patterns, constructed from dynamic memory access traces. Access graphs have memory objects for nodes, and their edges show how frequently the program accesses two objects together. Using access graphs, we reframe the memory layout problem as a combination of community detection and graph linear arrangement – both well researched problems with many good heuristic solutions. Based on these heuristics, we build a new algorithm called Hierarchical Memory Layouts (HML) that computes layouts with improved spatial locality. Hierarchical Memory Layouts combined with cache simulation give an estimate of possible cache improvement through layout changes. We use the output of HML to train random forest classifiers to automatically extract the relationships between data features (e.g. Cartesian coordinates) and memory access patterns (e.g. traversal). We then use the discovered relationships to improve the layout of the program data in memory or on disk.

The contributions of this work are as follows:

(1) Access graphs: A novel way of representing memory access patterns within a program. Access graphs reveal which objects in memory are accessed contemporaneously. They are computed from allocation and access traces. By using our proposed analysis techniques, programmers can automatically extract expert knowledge of access patterns that is key in most prior work on data layouts.
(2) Hierarchical Memory Layouts (HML): A new algorithm that uses access graphs to automatically derive improved data layouts for memory intensive programs. HML combines prior work in graph community detection and linear arrangement in a novel way in the context of spatial locality optimization. Using HML, programmers can get an estimate of how much room for improvement there is in a program's data layout. In certain workloads, HML can be used directly to recompute layouts of data in storage.
(3) Data-Driven Locality: A novel application of Random Forest Classifiers to detect correlations between memory access pattern and data properties. We use random forests to learn which features of data itself can be used at runtime to group allocated objects, with the goal of improving spatial locality. We use these techniques to automatically infer expert knowledge from prior work on data structure layouts (graphs and meshes), and expand the application to red-black trees. To evaluate the performance gains we built Tidy, a hint-based allocator wrapper that lets us use objects' data field values to guide allocation at runtime.

Figure 1 shows a detailed workflow diagram for the techniques presented in this paper. We will not go into much detail about each of the nodes in the diagram, but we encourage the reader to refer back to it as they read through sections 2-4. It is divided into three different stages, outlined with dotted lines. The first stage shows the process of *access graph creation*, which is described in detail in §2. The second stage is *layout performance evaluation*. In this stage, we evaluate the potential for performance improvement from changing the data layout of the program. Layout performance evaluation is covered in §3. Finally, if the programmer deems it worth to change the data layout based on cache simulation results, they proceed to the third stage – *Data-Driven Spatial Locality*, described in §4. In this stage, we use machine learning techniques to discover data features that can be used as hints for Tidy, our allocator wrapper.

The only parts of the workflow that require human input are the inspection of performance evaluation results and the modification to the program's allocator calls in the third stage. The automation of the performance evaluation is possible simply by setting predefined thresholds for cache miss improvement. We discuss the possibility of automating allocator call changes in §7.

The rest of the paper contains evaluation and discussion of our results in §5, a summary of related work in §6, envisioned future work in §7 and conclusion in §8.

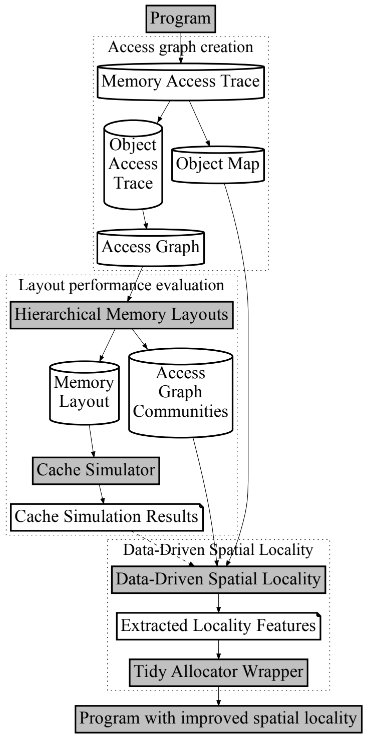

Figure 1: Workflow diagram

2 ACCESS GRAPHS

Access graphs are a novel way of aggregating a program's memory access trace to capture properties related to spatial locality. In this section we formally define access graphs and explain how we use them to reason about and improve spatial locality.

To refer to units of memory holding a datum of a specific type we interchangeably use the terms *data items*, *data elements* and *data objects*. The contents of the location could be an instance of a C struct, a C++ object or another kind of data – this distinction is not important for our tools.

Besides accesses to dynamically allocated (heap) objects, memory access traces contain accesses to global variables and local (stack) variables. We focus on large data structures that generate many cache misses – they are most likely to consist of dynamically allocated objects. Therefore, we filter data objects of other kinds from the trace.

Definition 2.1 (Object Access Trace). *Object Access Trace is a filtered form of a program's dynamic memory access trace. It is obtained by first removing all stack and global accesses from the trace. Next, all the accesses that target heap objects are replaced with accesses to the target objects' base addresses.*

Example: If an access writes to address `0xdeadbee8` and it is determined that the address is within the bounds of an object with base address `0xdeadbee0`, the write to `0xdeadbee8` is replaced with a write to `0xdeadbee0`.

Definition 2.2 (Object Map). *An Object Map is a hash map of all the allocated objects. It is generated by processing the original memory access trace, before converting it into an Object Access Trace. When an object is accessed, the offset at which the access was made is recorded in the map, along with the type of access (read/write) and the value read/written. Object Maps give information about the type of object and the contents of all of its data fields.*

Example: If an access writes the value 3.14 to address `0xdeadbee8` of the object at `0xdeadbee0`, the Object Map entry for that object is updated with information about a write with value 3.14 at offset 8.

Object maps allow us to connect memory addresses with properties of the corresponding objects. For example, the map may inform us that the memory address `0xdeadbee0` contains an object of type `mesh_node_t` with the value 12.34 at offset 0 (the x-coordinate field), and the value 42.1 at offset 8 (the y-coordinate field). The mapping between memory addresses and objects will be used later to find correlations between the access pattern and the object properties and to improve the data layout in memory or on disk. For example, our algorithms will automatically discover that objects with similar x, y coordinates are accessed close together in time; by allocating mesh objects with similar coordinates close together in space we will improve spatial locality and reduce the execution time.

Definition 2.3 (Access Graph). *An access graph is an undirected graph where there is a vertex for every dynamically allocated object in the Object Access Trace. Two vertices are connected by an edge if there are at least two contemporaneous memory accesses to the objects represented by these vertices. Two accesses are considered contemporaneous if they occur within C_L memory accesses of one another. C_L is called a locality constraint. Whenever we detect the first contemporaneous access we create an edge between the two vertices and assign it the weight of one. Whenever we detect another contemporaneous access to vertices already linked by an edge, we increment the edge's weight by one.*

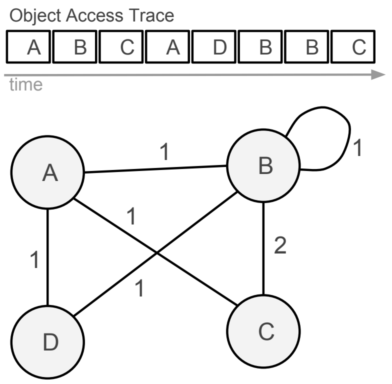

Figure 2: Access graph example

Figure 2 shows an example of an Object Access Trace with the corresponding access graph for $C_L = 2$. Note that D and C have no edge between them because there are no contemporaneous accesses to them in the trace. Objects B and C have an edge with the weight of two, because there are two contemporaneous accesses to them in the example trace.

Choice of C_L value for computing access graphs has two aspects that should be considered: *computational cost* and *captured information*. From the computational cost point of view, the C_L value tells us exactly how many edge additions/updates we have to perform for each access in the Object Access Trace. Because of this, C_L should be as low as possible, while retaining important information in the access graph. As for the second aspect, we empirically tested higher values for C_L for Hierarchical Memory Layout computation (described in §3), but did not observe a significant improvement in the quality of results. For our analysis techniques, we used $C_L = 2$, meaning we only consider two immediately adjacent accesses in the Object Access Trace. Using access graphs for other purposes may require reconsidering this choice.

From these definitions, we infer the following: *The weight of edges in an access graph tells us which objects get accessed contemporaneously the most. The more occurrences of accesses to A and B within a window C_L in the program's memory access trace, the heavier the weight of the edge between A and B.*

As a result, ***access graphs enable the automation of spatial locality optimizations that in the past, to the best of our knowledge, were performed manually***.

Given an access graph, how do we use it to improve spatial locality? The graph tells us which objects are accessed contemporaneously. How do we translate this information into a more efficent program? To answer this question, we will break it down into two parts. First, we will find out if there is a potential to reduce the cache miss rate by using a different data layout. Given an Object Access Trace and an Object Map, we will assume that we can rearrange the memory addresses of the objects in any way we like (without worrying how this could be achieved in practice), replay the access trace in a simulator and evaluate the cache miss rate resulting from the new layout. We compute the new layout using the new Hierarchical Layout Algorithm that we describe in §3. Although this is not a concrete solution that a programmer can use directly, evaluating layout changes in an abstract way will help us understand if there is a potential to improve performance by changing the layout.

The second part of the question asks how we can improve the data layout in a concrete program. Our work on Hierarchical Memory Layouts in §3 describes the process of obtaining good memory layouts that can be used directly to reorder data in storage. Section §4 presents a machine learning technique that trains on the Object Map and discovers the properties of the data objects available at object allocation time that can be used to guide object placement in memory at runtime.

3 HIERARCHICAL MEMORY LAYOUT

In an access graph, objects that have a lot of contemporaneous accesses are connected by heavily weighted, i.e., *strong*, edges. Relying on this property, we can reframe our grouping objective as a well-researched problem of *community detection in networks*.

Community detection algorithms detect groups of nodes in graphs such that the connectivity within a group is strong (many edges, higher weights), and the connectivity between groups is weak (few edges, lower weights). In the context of access graphs, community detection algorithms would place into the same groups objects that are often accessed contemporaneously, and would place into different groups objects that are rarely accessed contemporaneously.

Our Hierarchical Memory Layout algorithm extends a multilevel community detection algorithm by Blondel et al [4]. Multilevel community detection starts with every graph node representing its own community. It expands communities by adding close neighbours into them, making sure that the change will result in higher inter-cluster separation, and intra-cluster proximity (together called *modularity*). When such a change is impossible, the algorithm stores the community assignment and transforms the graph by fusing all nodes within a community into a single node, and aggregating all edges to other such community nodes. This process is repeated on the transformed graph until there are no changes that can be done that improve modularity.

Applying Blondel's algorithm on access graphs produces a hierarchy of communities. We will refer to the level with the highest number of small communities as the first level or bottom level, interchangeably. Subsequent levels with fewer larger communities will be referred to as being higher in the hierarchy.

Nodes in first level communities are not ordered internally. This may not be a problem if the entire community fits within a unit of spatial locality (e.g., cache line, VM page, disk page etc.). Unfortunately, we cannot choose the size of the communities; it is dictated by the access graph's structure. Frequently, first level communities turn out to be so large that a random permutation of objects within them loses any beneficial locality properties. In these cases we need to find a good internal ordering of objects for first level communities.

Ordering access graph nodes within first level communities has the following rules:

- The stronger the edge between two objects, the closer they should be placed in the layout
- Relative placement of object pairs that have no edge between them is irrelevant.

These rules are in line with the optimization objective of the Minimum Linear Arrangement problem (MinLA). Minimum Linear Arrangement is a known NP-hard problem, for which researchers have explored many heuristics [26]. Because access graph edges are weighted, we use the weighted variant of the problem.

Definition 3.1 (Weighted Minimum Linear Arrangement). *Given a graph*
$G(V, E)$, $|V| = n$,
find a one-to-one function
$\varphi : V \longrightarrow \{1, .., n\}$
that minimizes the Linear Arrangement cost (LA), defined as
$LA(G, \varphi) = \sum_{(u,v)\in E} |(\varphi(u) - \varphi(v)) * w|$

Definition 3.1 states that Minimum Linear Arrangement has the objective of linearly laying out graph nodes so that it minimizes the distance between connected nodes. The weighted version of the problem prioritizes reducing the distance between pairs of nodes with stronger edges. In the context of access graphs, MinLA heuristics will try and place objects that are frequently accessed contemporaneously as close as possible in the memory layout.

In our work we use the Spectral Sequencing [16] heuristic proposed by Juvan et al. to approximate solutions to MinLA on access graphs' communities.

Definition 3.2 (Spectral Sequencing). *Spectral Sequencing computes the Fiedler vector of the graph G – the eigenvector $x^{(2)}$ corresponding to the second lowest eigenvalue λ_2 of the Laplacian matrix L_G of the graph G.*
It then produces the ordering function φ such that
$\varphi(u) < \varphi(v) \Leftrightarrow x^{(2)}(u) < x^{(2)}(v)$

Spectral Sequencing was shown [26] to give results of good quality, at a low computational cost. Hierarchical Memory Layouts use Spectral Sequencing as a sub-algorithm, but it can be replaced with any suitable MinLA heuristic.

The Hierarchical Memory Layout algorithm operates on the Access Graph (constructed from the Object Access Trace) in two phases, utilizing the two previously described algorithms.

The first phase performs multilevel community detection, producing community levels $L_1, ..., L_n$, where L_1 represents the first computed community level – one with the largest number of small communities. As the levels increase, communities become fewer in number, and greater in size.

The second phase performs Spectral Sequencing on each community in L_1. The objects within each L_1 community are ordered according to the linear arrangement obtained from Spectral Sequencing. Every community level contains all of the nodes in the original graph – the only difference is how the nodes are grouped.

Our use of both community detection and Minimum Linear Arrangement heuristics begs the question: Why not use Spectral Sequencing to lay out the entire access graph? This is a valid question, and using only Spectral Sequencing would produce good layouts. However, a linear layout of nodes in a graph obscures a property that is needed for our data-driven spatial locality technique. Data-driven spatial locality needs *groups* of data objects to use as training class labels (the whole process is described in detail in §4). Community detection algorithms output groups with desirable properties – strong connections within a group, and weak connections to nodes in other groups.

To construct the final layout, we label each object with a *community vector*. Community vector of an object is a set of indices $(I_n, I_(n-1), ..., I_1, I_{SS})$. Index I_k is simply a unique identifier for the community at level k that the object belongs to. I_{SS} is the linear layout index of the node within its L_1 community. Due to the nature of Blondel's multilevel community detection algorithm, if two nodes have the same I_k index, they are guaranteed to have the same I_{k+1} index.

We lexicographically sort the objects by their community vectors to produce the final Hierarchical Memory Layout.

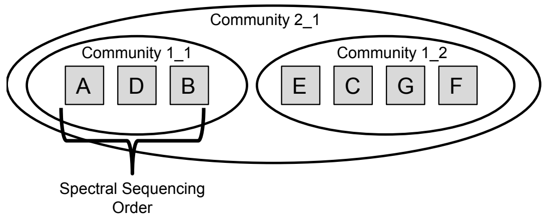

Figure 3: Hierarchical Memory Layout example

Figure 3 shows a simple example of the information produced by Hierarchical Memory Layouts. The two detected first level communities ({A, D, B} and {E, C, G, F}) are internally ordered by Spectral Sequencing. They belong to the same second level community and will be placed next to each other in the final layout.

We use the output of Hierarchical Memory Layout for two purposes. First, we rearrange the objects in the memory access trace according to the layout and feed the new trace into the cache simulator (§5.1) to estimate whether improving data layout is likely to improve performance. Second, we train a machine learning model (§4) to find out if any data features of the objects can be used during data allocation phase to improve performance in a concrete program.

4 DATA-DRIVEN LOCALITY

Prior work on memory layouts uses expert domain and algorithm knowledge to obtain better layouts. These solutions often use features of the data itself, such as object fields, to inform the generation of layouts. For example, in mesh traversals, the visitation order goes from one mesh object (triangle, point, edge) to its neighbours. Thus, grouping these objects in memory based on their spatial proximity (neighbours are close together in Euclidean space) yields layouts that exhibit better spatial locality. Another example are iterative graph algorithms such as PageRank. Solutions like GraphChi [17] group edges by source and destination nodes to improve spatial locality. However, the process of deriving these layouts is largely manual and relies on expert knowledge in both the algorithm's domain and memory optimization.

We present a way to automate the discovery of correlations between spatial locality expressed in the algorithm and the features of data itself.

The question our technique aims to answer is the following:

Given the Object Map described in §2, and first level communities detected by Hierarchical Memory Layout algorithm, is it possible to decide which community an object belongs to, based only on its data features?

We use random forest classifiers [5] for this task. Random forests are a learning method for classification and regression that utilizes a combination of multiple decision trees and overcomes the decision trees' tendency to overfit. We chose random forests for two main reasons:

(1) They give good results and are relatively easy to set up compared to other classifiers such as neural networks.
(2) They are less opaque than other techniques. This means that it once a random forest learns to classify data from a dataset, it is possible to extract the contribution of input vector elements to the decision process (further explained in §4.2).

However, our technique is not inherently tied to random forests. It can use any other suitable classification algorithm without modification.

4.1 Generating input vectors

The input vectors for our classifier are generated from the set of all detected data features. We split data features into three categories: primary features, secondary features and meta-features.

The first, trivial, type of data features are primitive data fields - *primary features*. Primary features are all non-pointer fields within an object. An example of this would be the coordinates of an object in a mesh.

The second type of data features are primitive features of neighbouring objects - *secondary features*. Here, object A is a neighbour of object B if B contains a pointer field that holds the value of A's memory address. Secondary features can be used in allocation policies when the neighbouring objects are initialized and known in advance. For example, if the mesh traversal workload's triangle object is written so that it only contains pointers to the triangle's points, we can use the points' Euclidean coordinate features to inform the allocation of triangles (provided points are initialized before triangles).

The third type of data features are meta-features. These are not present in the data itself as object fields, but rather describe some inherent properties of objects. Examples of meta-features are array indices of objects, memory addresses assigned to objects within the observed execution trace, size/type of object, allocation point in the source code, etc. A correlation between spatial locality and array index (or memory address) of objects can indicate that the current layout already does well in terms of spatial locality. A correlation between size and spatial locality would mean that one should use an allocator that bins objects based on allocation size (a common strategy [19] [11]).

4.2 Training methodology and evaluation criteria

Our full dataset consists of all the objects in the access graph, with their input feature vectors and community labels. When training the classifier, we split the full dataset into randomly picked 80% / 20% subsets. The 80% partition is used for training, and we verify and compute accuracy on the remaining 20%.

To extract the features that contribute the most to the accuracy (feature importance), we use the *gini method* proposed by Breiman [6], implemented in scikit-learn's [23] `RandomForestClassifier` class. This method evaluates a feature's importance as the measure of all decision tree splits that include the said feature, normalized over the entire forest. The more decision tree splits a feature is involved in, the more important it is deemed for the classifier. Categorical accuracy is the percentage of samples in the test dataset for which the classifier predicted the correct label. Top-5 categorical accuracy is the percentage of samples in the test dataset for which the correct label was within the top 5 choices of the classifier. We report categorical accuracy, top-5 categorical accuracy and feature importance information in §5.3.

4.3 Tidy: a memory allocator wrapper

To test the impact of using detected data features to inform layout at runtime, we built Tidy. Tidy is an arena-based allocator, meaning that it organizes allocated objects into different arenas. Unlike most other arena-based allocators, Tidy chooses the arena for the newly allocated object based on a hint provided by a programmer. In our context, the hint is the feature of the object that correlates with the desired object grouping – the feature that we discover using random forests. The idea is that upon object allocation the programmer would pass to the allocator wrapper the value of that feature. For example, if the programmer is allocating a vertex of a triangle, she would pass the X and Y coordinates to Tidy. Tidy would then convert these values into an arena index, such that vertices with similar X, Y coordinates would get allocated in the same arena. Using this method we achieve the desired grouping of objects, the one suggested to us by the access graph, in a concrete execution.

The idea of hint-based allocators is not new. Chilimbi et al. [9] propose `ccmalloc` – a cache conscious allocator that accepts hints. Hints in `ccmalloc` are addresses of previously allocated objects; the allocator attempts to place the new item as close as possible to the one whose address was provided as the hint. Tidy can be adapted to use different kinds of data for hints, and we consider it a generalization of the `ccmalloc`'s approach.

The hint taken by Tidy has a form of an n-dimensional vector; the size of the vector is given to Tidy upon initialization and is stored in a Tidy context. Tidy then allocates an n-dimensional array of arenas, and the vector elements (modulo the size of the dimension) will be used to index into that array. This implies a linear mapping of hints to the arena space. Non-linear mappings are also possile; we plan to explore them in the future.

To use Tidy, the programmer needs to replace calls to existing memory allocators with calls to Tidy. If a call to a standard `malloc` routine has the interface of:

```
malloc(size_t size);
```

a call to Tidy looks like:

```
tidy_alloc(tidy_ctx_t *ctx,
           size_t size,
           unsigned int *hint);
```

The programmer can configure the size of the arena as well as the size of the dimension, or opt to use the default settings. More experiments are needed to determine whether an optimal arena size can be pre-determined from the properties of the access graph, if it needs to be tuned individually for the workload or if there is a single (perhaps architecture-dependent) default that works well across the board.

The programmer needs not specify the total number of arenas in advance or the total number of allocated elements; if Tidy runs out of space in an arena, it allocates a new one for the same set of hints. To allocate arenas, Tidy uses `libc malloc`, but it can be changed to use any other allocator.

5 EVALUATION

In this section we first show how Hierarchical Memory Layouts (§3) can be used to estimate potential performance improvements from improved data layouts (§5.1). Following that, in §5.2 we show how HML can be used directly to derive better data layouts in storage. Finally, we show that data-driven layout techniques (§4) can be used to detect correlations between data features of objects and their layout, where such correlations exist, and guide dynamic memory allocations.

In our experiments we use nine applications, two of which are used to evaluate improved storage layout only. .

Simple data structure benchmarks. The three benchmarks in this set are our own implementations of PageRank, mesh traversal and red-black trees in C/C++.

PageRank stores data as node and edge objects. The graph is initialized from an edge list file. Each node and edge is separately allocated using C++'s `operator new()`. Nodes contain their ID, the data field and vectors of pointers to their in-edges and out-edges. Edges contain the IDs of the source and destination nodes, and a floating point data field.

Mesh traversal operates on a 2D network of node and triangle objects. Triangles contain pointers to their three nodes, and three adjacent triangles. Nodes contain their x and y coordinates and a vector of pointers to all adjacent triangles. The data is initialized by allocating objects one by one, according to input from a node and triangle list file.

Red-black trees are collections of nodes, where each node has pointers to its parent and two children, a floating point payload, and a colour field. The benchmark fills the tree with random nodes, and then executes a series of lookups.

SPEC CPU2017 memory intensive benchmarks. SPEC CPU2017 is the 2017 release of the popular benchmark suite. For our experiments, we used three memory-intensive benchmarks (according to Amaral et al.[2]): 505.mcf, 520.omnetpp and 531.deepsjeng. 505.mcf is a mass transportation route planning program written in C. 520.omnetpp is a discrete event simulator of a large 10 gigabit network, written in C++. 531.deepsjeng is a speed-chess program with deep positional understanding, written in C++. Each manipulates a large number of heap objects, making them suitable for applying our Hierarchical Memory Layout algorithm.

Kyoto Cabinet's kcstashtest. Kyoto Cabinet is a key-value database management library. It is a direct successor of Tokyo Cabinet, developed by FAL Labs and used by Japanese social network Mixi. It contains multiple different implementations of the data store back-end. The benchmark shown here, kcstashtest, performs writes and reads in Kyoto Cabinet's StashDB data store variant. StashDB internally keeps records in hash tables.

Graph traversal benchmarks. To evaluate how using HML can improve data layouts in storage, we use graph traversal benchmarks of our own implementation.

We obtain the memory access traces required for generating the access graphs using DINAMITE [22]. DINAMITE is an LLVM pass that instruments every memory access and compiles the program, such that information about memory allocations, accesses, and data types is emitted to a log file. Our techniques would work with any memory access tracing tool that provides this information.

To estimate the performance effect of changing the data layout via HML, we simulate a cache hierarchy using Dinero IV[13]. The simulated Dinero cache is modelled after Intel® Core^{TM} i5-7600K. L1 D-cache has the capacity of 64kB, and is 8-way set associative. L2 has the capacity of 256kB, and is 4-way set associative. LLC has the capacity of 8MB, and is 16-way set associative. Our DTLB simulator has 128 4kB page entries, and is 4-way associative.

5.1 Hierarchical Memory Layouts

The main purpose of our Hierarchical Memory Layout algorithm is to provide an *estimate of the upper bound on performance improvement from changing the data layout.* The output of the algorithm are multilevel communities produced by the first stage described in §3 and the final layout which maps original object addresses to addresses of the objects in the improved layout.

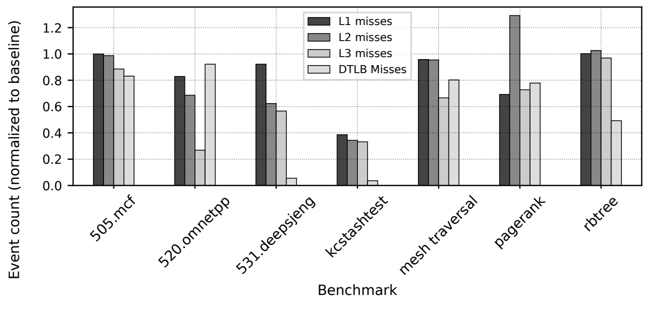

Figure 4: Hierarchical Memory Layouts cache and DTLB misses. Event counts are normalized to original layout results.

Figure 4 shows simulated cache event counts for the first seven benchmarks after applying the Hierarchical Memory Layout algorithm. The numbers are normalized to the simulated cache event counts of the original layout. Five benchmarks out of seven show significant improvements in cache miss rates.

We notice a consistent trend: HML improves the miss rate at the higher levels of the memory hierarchy, such as the last-level cache (LLC) and the TLB, to a larger extent than at the lower level of the hierarchy (L1 and L2). That is because it is easier to organize objects in larger groups accessed contemporaneously within a relatively large time window (macro grouping) than into small groups accessed contemporaneously within a very short time window (micro grouping).

Furthermore, performance improvements depend on the size of data objects. In 505.mcf, for example most of the allocated objects are over 100B in size and do not fit into a single cache line. Thus, performance improvements that could occur due to more efficient packing of objects within the same cache line do not happen.

From the red-black tree benchmark we learn that the extent of improvements from HML also depends on the access pattern. In red-black trees, the allocated objects are not as big as the ones in 505.mcf, but the tree itself exceeds the cache memory capacity by far. The combination of the large dataset and random lookups means that the algorithm does not revisit the same tree nodes often. The improvement in cache performance is thus low. However, DTLB misses improve by 51%.

Our conclusion is that the concrete performance gains from HML depend largely on the size of objects, size of the working set and the access pattern of the program. Furthermore, HML tends to better improve spatial locality at a coarser granularity: at the level of the TLB or the LLC.

Let us look back on what these HML results mean. Our algorithm operates under the assumption that it is possible to reorder objects in memory in an arbitrary way. This assumption does not hold for the majority of real-world programs. Recorded memory access traces, on the other hand, are an idealized environment for testing different layouts. The main purpose of HML is to give an estimate of how much performance is to be gained from changing the layout of items in the best case. We show that for the selected memory-intensive benchmarks there is much room for performance improvement from reordering data.

We explore two ways in which output from Hierarchical Memory Layouts can be used in practice to achieve better performance in programs. In §5.2 we explore the possibility of using HML to directly inform the layout of data in storage, and in §5.3 we show the results of applying data-driven layout techniques (§4) to our benchmark set.

When such optimizations are not possible in practice, Hierarchical Memory Layouts provide a starting point for work on layout improvement. The output of HML is a concrete layout of data that improves spatial locality, groups of objects that get accessed frequently together within the given program, and a descriptive Object Map which ties the previous two to the actual data within the program. Researchers in the future can use these layouts as stepping stones towards new locality optimization techniques.

5.2 Data layout in storage

Programs whose data layout is directly inherited from an input file, for example those that `mmap` the input file to materialize data in memory or those that dynamically allocate data objects in the same order as they appear in the input file, can directly benefit from the HML technique. We can reorganize the input data in the file in the same order as suggested by the HML algorithm and as a result obtain better spatial locality at runtime.

To evaluate such a scenario we wrote an application in C++ that performs graph traversal using either breadth-first search (BFS) or depth-first search (DFS) order. We ran two benchmarks. The first one performs ten BFS traversals starting from a randomly selected code each time. The second benchmark works the same way, but uses DFS.

For these benchnmarks, an optimal strategy for spatial locality would be to allocate the nodes in the same order as they are traversed, but because the traversal begins with a different node each time, there is not a single "optimal" layout that we can use. Instead, we run the HML algorithm on the memory access traces for these benchnmarks to suggest an improved layout. HML outputs the order of the nodes, where each node is identified by its unique ID. We then reorganize the input file such that the nodes appear in the same order as suggested by HML. We create one input file for the BFS benchmark and another one for DFS.

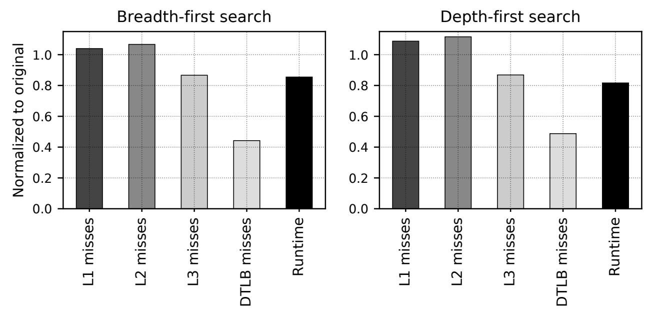

Figure 5: Cache misses, TLB misses, and runtime for HML-derived layouts in graph traversals. Normalized to original layout metrics.

Figure 5 shows the runtime, cache misses and TLB misses obtained using the HML, relative to the baseline layout, where nodes in the input file are sorted by their numeric ID. These measurements were obtained on the actual hardware. We do not provide simulation results, because we are able to apply HML directly. We observe an improvement in runtime of 14% for the BFS and 18% for the DFS. L1 and L2 miss counts show a slight degradation, which is made up for by a significant reduction in L3 and DTLB misses. Again, we see the pattern observed in §5.1, where HML tends to optimize better for L3 and TLB, while keeping L1 and L2 cache misses the same or slightly worse than the original layout. The improvements in L3 and DTLB misses outweigh this degradation and produce a positive effect on the running time.

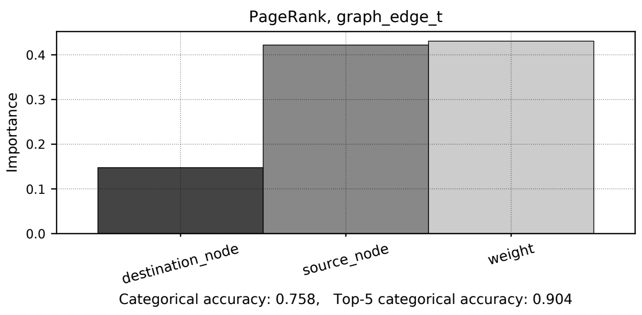

Figure 6: Data feature importance and categorical accuracies, PageRank

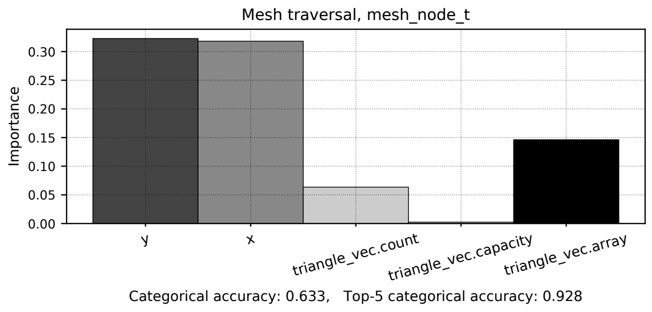

Figure 7: Data feature importance and categorical accuracies, mesh traversal

5.3 Data-driven locality

We applied the data-driven locality techniques to the first seven benchmarks described in §5. In three of these, *mesh traversal*, *PageRank* and *red-black trees*, our system identified data features that could be used as hints for the Tidy memory allocator.

For PageRank and mesh traversal, our system automatically identified the same features that in the past were discovered manually: source nodes for *PageRank* [17] and Cartesian coordinates for *mesh traversal* [33]. This was a positive confirmation of the effectiveness of our techniques.

Figure 6 shows that the source node is the main contributor to accurate community prediction in PageRank. The edge weight also has high predictive power, but it is not available at runtime, so we disregard it when testing new layout strategies with Tidy.

Figure 7 shows importance scores of mesh node fields in the mesh traversal algorithm. We can see that the x and y Cartesian coordinates were picked up by the random forest as being the most important for classification. The community prediction accuracy is high, meaning we can use the x and y coordinates to group objects at allocation time with Tidy.

Red-black trees are considered difficult to optimize for spatial locality, and we are not aware of any heuristics used in the past to improve their layout. Our system, on the other hand, was able to discover one. Figure 8 shows that the payload field, which is used to rebalance the tree, has the highest predictive power. Right behind it are the child node payloads, which makes intuitive sense because they are directly related to the current node's payload. With categorical prediction accuracy of 0.98, red-black trees have the features with the strongest predictive power of all three benchmarks.

We used the discovered fields in all three benchmarks to generate hints for Tidy allocator wrapper. Our mapping is relatively simple, performing one or two-dimensional binning. We bin values into buckets by dividing the value space into a grid. Each grid cell corresponds to one integer hint value. The grid size was experimentally tuned to the best performing value, and the performance numbers on real hardware are reported in Figure 9.

The results of our Tidy experiments are in line with the results from the simulated evaluation presented in §5.1. We see a runtime improvement of 25% for mesh traversal, 27% for PageRank and 14% for red-black tree queries. These improvements correlate with the reduction in cache/DTLB misses. We observe the same trend we saw in simulation – grouping items with Tidy does better for memories with higher latencies, in case of red-black trees even degrading L1 and L2 miss counts by 20%. As we observed earlier, HML is better at macro grouping than at micro grouping, providing improvements at the higher level of the memory hierarchy (L3 and TLB), but not necessarily at the lower level. This will sometimes result in L1 and L2 cache miss degradation, but L3 and DTLB improvements typically outweigh these losses.

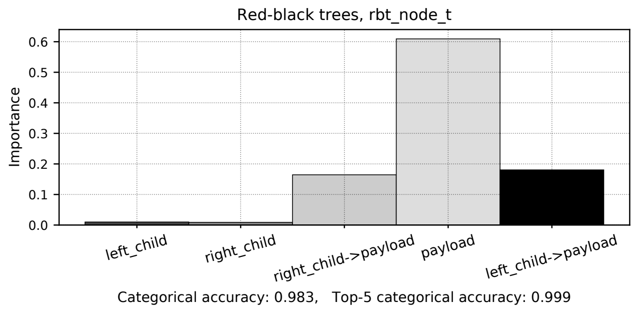

Figure 8: Data feature importance and categorical accuracies, red-black trees

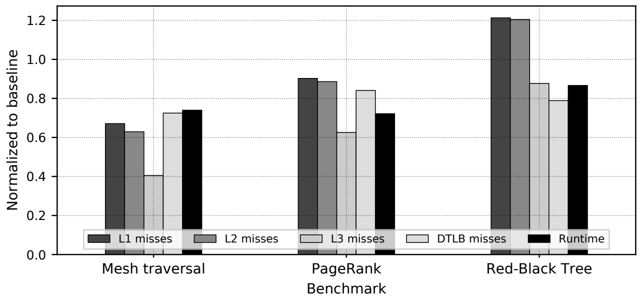

Figure 9: Performance improvement from using Tidy allocator wrapper with hints based on the knowledge extracted by random forests

Discrepancies in numbers between the simulation and Tidy results can be attributed to two factors. First, Tidy is a best-effort layout strategy. It opportunistically allocates objects within the same memory buckets without ordering them in the exact same way as HML would. This is a limitation of using dynamic allocation – we cannot expect the program to anticipate the exact place in memory where each object should be stored. Second factor is the imprecision in the simulator itself. We use Dinero IV, configured to mimic the caches of our test hardware, however, we have no way of knowing how it differs from the hardware itself.

6 RELATED WORK

Cache-conscious structure layout and definition [8] [9] blazed the trail for the ideas presented here. Collocating contemporaneously accessed objects and hint-based allocators (the previously discussed `ccmalloc`) were both explored by the authors. The access graphs allow for novel analyses that help programmers *understand which data should be grouped at runtime.*

Profile-based data layout optimizations were explored by Rubin et al. [28] Their approach uses profiling techniques to identify objects that are top offenders to cache performance. This information is used to apply a series of known layout optimizations, such as structure splitting, field reordering and reordering of whole objects. Our work attempts to solve the layout problem in a more holistic fashion – by understanding the interplay between accesses to all of the objects in a data set.

The use of data structure fields to inform data layout creation was inspired by GraphChi [17]. The authors proposed a layout specifically for bulk synchronous processing on graphs; we aspired to capture the essence of these ideas, so they could be used for data structures in general. With access graphs, we aimed to remove the necessity of domain knowledge for reasoning about good layouts. We showed that our techniques reach similar conclusions about laying out edges for Pagerank.

Higher order theory of locality (HOTL) [32] sets up a mathematical framework for thinking about different locality metrics. From it, Xiang et al. develop a novel low-overhead way of locality sampling, and demonstrate the ability to predict miss rates from the acquired information. While providing an excellent basis for reasoning about locality, techniques presented in HOTL do not expose actionable data on how to improve locality in a program. Access graphs take a different approach of observing the full access trace and providing insight into how objects relate to each other, in terms of access locality. This level of detail, while incurring large overheads compared to sampling techniques, brings new information on what can be done to data layout to improve performance.

Yoon et al. [33] use a graph representation for generating a cache-oblivious layout of the mesh, but their representation is very different from locality graphs. Edges are formed between vertices connected in a mesh; in other words this representation is specific to the mesh data structure and requires knowing which vertices are connected. Locality graphs operate on any generic memory access trace and require no knowledge of data structure specifics.

Liu et al. propose a sampling tool [20] that correlates bad memory performance with data objects: static or dynamically allocated variables. Similar approaches can be found in DProf [25] and MemProf [18], which correlate cache misses and remote memory accesses to data items. Looking at data as the first order citizen in memory performance analysis is a good approach to helping programmers better understand memory bottlenecks. Access graphs use a similar data-centric approach, but aspire to bring more actionable insight by observing the entire execution trace and extracting locality-related relationships between data objects.

Peled et al. propose a context-based cache prefetcher model [24], that detects *semantic locality* and issues prefetches based on it. Their approach is to implement a machine learning model in hardware that observes "contexts" of memory accesses during execution. A context contains information such as register contents, access and branch histories, compiler provided type information, etc. The approach used by Peled et al., and our data-driven spatial locality method are similar in that they both base their techniques on data-centric contextual information. They are on two sides of the trade-off spectrum: semantic locality is generally applicable out of the box and has insight into lower-level information, but the amount of data it can observe at any point in time is limited due to hardware constraints. Access graphs and their related tools observe a much larger body of information about the execution, but at a higher level. They do not provide performance benefits on-the-fly, but is used to derive better locality optimizations.

Previous allocator work such as dlmalloc [19] and jemalloc [11] inspired Tidy's arena-based allocation. However, these allocators rely on the information available through regular allocation calls, namely the size of the object being allocated, to group data. Our approach with Tidy was redefining what an allocation call looks like, and showing that additional information can further improve locality. We do not compare against dlmalloc or jemalloc, as our work is mostly orthogonal – we show that informed hints can improve a layout's performance even when allocating objects of the same size throughout the program. Tidy is a proof of concept allocator wrapper, and our implementation does not focus on the allocation speed. The insights from Tidy can be used to bring hint-based allocation into state-of-the-art allocator libraries.

7 FUTURE WORK

In future work, we plan to explore the possibilities of using access graphs and Hierarchical Memory Layouts for different kinds of optimizations. In its current state, the HML algorithm optimizes spatial locality at a relatively coarse granularity – the entire data set. As we have seen in §5, this sometimes results in degradation in smaller L1 and L2 cache performances. We would like to explore the possibility of combining different techniques to consistently gain performance over the entire cache hierarchy. Taking the concept of locality a step further, we are interested in exploring the possibility of using access graphs and HML to organize data over multiple machines in a distributed environment.

We also plan to address the automation of changing allocator calls to Tidy. This task can be split in two parts: *changing the allocator calls in the program* and *automatically finding a good mapping function that translates data features into allocator hints*. The first part is engineering and can be tackled either from the compiler level (writing an LLVM pass) or by changing the plain text source code directly. The second part is a complex research question. One approach would be to explore the relationship between detected community sizes and the values their objects' data. A different approach could be embedding the random forest classifier directly into Tidy and using it at runtime. The main concern here is the impact of adding a fully trained random forest on performance of allocation calls. With the advent of hardware such as Google's TPU cores, it may be possible to use hardware-accelerated neural networks for this task.

8 CONCLUSION

In this paper, we present a novel data-driven approach to spatial locality. We introduced access graphs – our new abstraction of memory access patterns. Access graphs capture the information about which objects in a programs data set get accessed together frequently. We show that our novel Hierarchical Memory Layout (HML) algorithm can estimate the potential for cache performance improvement based on access graphs. We demonstrate how Hierarchical Memory Layouts can be used in practice, not only as a performance estimation tool, but to directly generate better data layouts in storage. Finally, we show a machine learning technique that discovers features of data itself that can be used as hints for Tidy – our hint-based allocator wrapper. We support our research with measurements that show significant performance improvements over a set of benchmarks.

REFERENCES

[1] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. 1999. DBMSs on a modern processor: Where does time go?. In *VLDB" 99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*. 266–277.

[2] José Nelson Amaral, Edson Borin, Dylan Ashley, Caian Benedicto, Elliot Colp, Joao Henrique Stange Hoffmam, Marcus Karpoff, Erick Ochoa, Morgan Redshaw, and Raphael Ernani Rodrigues. [n. d.]. The Alberta Workloads for the SPEC CPU 2017 Benchmark Suite. ([n. d.]).

[3] Michael A Bender, Erik D Demaine, and Martin Farach-Colton. 2000. Cache-oblivious B-trees. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE, 399–409.

[4] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.

[5] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[6] Leo Breiman. 2002. Manual on setting up, using, and understanding random forests v3. 1. *Statistics Department University of California Berkeley, CA, USA* 1 (2002).

[7] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. 2002. Cache oblivious search trees via binary trees of small height. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 39–48.

[8] Trishul M Chilimbi, Bob Davidson, and James R Larus. 1999. Cache-conscious structure definition. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 13–24.

[9] Trishul M Chilimbi, Mark D Hill, and James R Larus. 1999. Cache-conscious structure layout. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 1–12.

[10] Erik D Demaine. 2002. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets* 8, 4 (2002), 1–249.

[11] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada*.

[12] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 37–48.

[13] Mark D Hill and J Elder. 1998. DineroIV trace-driven uniprocessor cache simulator.

[14] Martin Isenburg and Peter Lindstrom. 2005. Streaming meshes. In *Visualization, 2005. VIS 05. IEEE*. IEEE, 231–238.

[15] Martin Isenburg, Yuanxin Liu, Jonathan Shewchuk, and Jack Snoeyink. 2006. Streaming computation of Delaunay triangulations. In *ACM transactions on graphics (TOG)*, Vol. 25. ACM, 1049–1056.

[16] Martin Juvan and Bojan Mohar. 1992. Optimal linear labelings and eigenvalues of graphs. *Discrete Applied Mathematics* 36, 2 (1992), 153–168.

[17] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a pc. USENIX.

[18] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. 2012. MemProf: a memory profiler for NUMA multicore systems. In *ATC-USENIX Annual Technical Conference*.

[19] Doug Lea. 2010. Dlmalloc.

[20] Xu Liu and John Mellor-Crummey. 2011. Pinpointing data locality problems using data-centric analysis. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. IEEE, 171–180.

[21] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent Data Structures for Near-Memory Computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 235–245.

[22] Svetozar Miucin, Conor Brady, and Alexandra Fedorova. 2016. DINAMITE: A modern approach to memory performance profiling. *arXiv preprint arXiv:1606.00396* (2016).

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[24] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. 2015. Semantic locality and context-based prefetching using reinforcement learning. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 285–297.

[25] Aleksey Pesterev, Nickolai Zeldovich, and Robert T Morris. 2010. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems*. ACM, 335–348.

[26] Jordi Petit. 2003. Experiments on the minimum linear arrangement problem. *Journal of Experimental Algorithmics (JEA)* 8 (2003), 2–3.

[27] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 472–488.

[28] Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. 2002. An efficient profile-analysis framework for data-layout optimizations. In *ACM SIGPLAN Notices*,

Vol. 37. ACM, 140–153.
[29] Pedro V Sander, Diego Nehab, and Joshua Barczak. 2007. Fast triangle reordering for vertex locality and reduced overdraw. In *ACM Transactions on Graphics (TOG)*, Vol. 26. ACM, 89.
[30] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, Vol. 48. ACM, 135–146.
[31] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.
[32] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: a higher order theory of locality. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 343–356.
[33] Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha. 2005. Cache-oblivious mesh layouts. In *ACM Transactions on Graphics (TOG)*, Vol. 24. ACM, 886–893.
[34] Sung-Eui Yoon and Dinesh Manocha. 2006. Cache-Efficient Layouts of Bounding Volume Hierarchies. In *Computer Graphics Forum*, Vol. 25. Wiley Online Library, 507–516.