A Unifying Abstraction for Data Structure Splicing

Louis Ye
University of British Columbia
Vancouver, BC, Canada
louisye@ece.ubc.ca

Mieszko Lis
University of British Columbia
Vancouver, BC, Canada
mieszko@ece.ubc.ca

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

ABSTRACT

Data structure splicing (DSS) refers to reorganizing data structures by merging or splitting them, reordering fields, inlining pointers, etc. DSS has been used, with demonstrated benefits, to improve spatial locality. When data fields that are accessed together are also collocated in the address space, the utilization of hardware caches improves and cache misses decline.

A number of approaches to DSS have been proposed, but each addressed only one or two splicing optimizations (e.g., only splitting or only field reordering) and used an underlying abstraction that could not be extended to include others. Our work proposes a single abstraction, called *Data Structure Access Graph* (D-SAG), that (a) covers all data-splicing optimizations proposed previously and (b) unlocks new ones. Having a common abstraction has two benefits: (1) It enables us to build a single tool that hosts all DSS optimizations under one roof, eliminating the need to adopt multiple tools. (2) It avoids conflicts: e.g., where one tool suggests to split a data structure in a way that would conflict with another tool's suggestion to reorder fields.

Based on the D-SAG abstraction, we build a toolchain that uses static and dynamic analysis to recommend DSS optimizations to developers. Using this tool, we identify ten benchmarks from the SPEC CPU2017 and PARSEC suites that are amenable to DSS, as well as a workload on RocksDB that stresses its memory table. Restructuring data structures following the tool's suggestion improves performance by an average of 11% (geomean) and reduces cache misses by an average of 28% (geomean) for seven of these workloads.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; • **Software and its engineering**; • **Computer systems organization**;

KEYWORDS

Memory performance, data structure design and analysis, CPU cache

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MEMSYS '19, September 30-October 3, 2019, Washington, DC, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7206-0/19/09…$15.00
https://doi.org/10.1145/3357526.3357548

ACM Reference Format:
Louis Ye, Mieszko Lis, and Alexandra Fedorova. 2019. A Unifying Abstraction for Data Structure Splicing. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19), September 30-October 3, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3357526.3357548

1 INTRODUCTION

Hardware caches rely on two kinds of intuition about locality: *temporal* locality (i.e., that recently accessed data is likely to be accessed in the near future), and *spatial* locality (i.e., that data placed together in the address space are also likely to be accessed together in time).

To take advantage of spatial locality, they fetch data from memory in batches ("cache lines" of usually 64–128 bytes). If the program has a high degree of spatial locality, hardware caches are operating efficiently: once they fetch (or pre-fetch) a cache line, all or most of the line will be used by subsequent accesses. Efficient caching lowers the memory access latency and reduces memory bandwidth requirements, and results in better performance.

Unfortunately, spatial locality in modern applications is very low [14]. Figure 1 shows that the cache line utilization of different benchmarks — i.e., the number of bytes accessed out of the 64 bytes brought into the cache — is only about 40% of the cache line on average. The appetites of modern datacentre workloads already

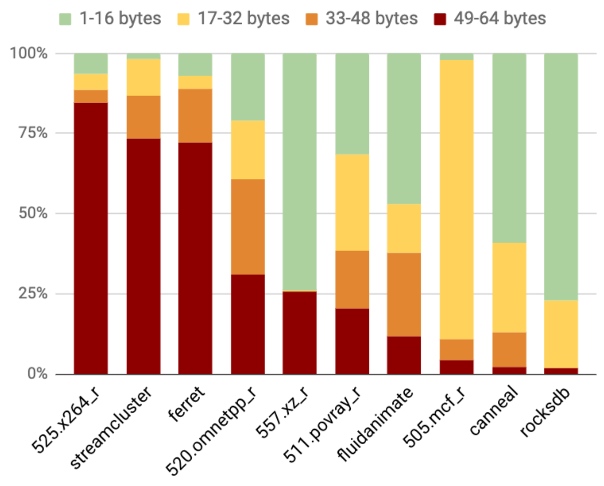


Figure 1: Bytes used in a cache line before eviction (64-byte cache lines, 8MB LLC, 16-way set associative), measured in an enhanced version of the DineroIV [9] cache simulator. In most applications, very little of a cache line is used before the line is evicted.

exceed the capacities of feasible caches [11], and low utilization only exacerbates this problem.

Spatial locality can be improved either by reorganizing the cache structure (in hardware) or by reorganizing the data layout (in software). While hardware solutions have been proposed (e.g., adaptive cache line granularity [e.g., 14] or sub-cache-line filtering [e.g., 22]), they have so far not found implementation in processors from the major commercial vendors. In this work, therefore, we assume fixed hardware and focus on restructuring the application's data layout.

Writing programs with good spatial locality is hard. Data structures are often naturally organized along primarily semantic lines, and developing intuition about co-temporal accesses to single fields in a large structure is rarely straightforward. To help programmers with this challenge, prior work proposed tools that recommend (or automatically make) changes in data structure or class definitions. Examples include *class splitting* — when "hot" or frequently access fields are segregated from "cold" or infrequently accessed fields [7, 13, 16]; *field reordering* — when fields that are accessed together are co-located in the data structure [7, 10]; and *pointer inlining* — when a pointer in a structure is replaced with the data it refers to avoid locality-breaking indirection [8].

We observe that all these techniques, which we call *data structure splicing* (DSS), are conceptually similar. They rely on reorganizing the definitions of classes or data structures guided by observations of how individual fields are accessed. Yet, there isn't a common, unifying abstraction that enables reasoning about these optimizations. Lack of a common abstraction has several downsides. First, different optimizations were embodied in disparate tools; to apply all of them, the developer may need to use, for example, one tool for class splitting, another one for field merging and yet another one for pointer inlining. Second, disparate tools may produce conflicting recommendations: for example, one tool may suggest to reorder fields in a way that conflicts with a class-splitting suggestion produced by another tool. Third, other conceptually similar optimizations, such as class merging and field migration, as we show in this paper, were overlooked in prior work.

Our main contribution is the unifying abstraction for data structure splicing. The abstraction is the *Data Structure Access Graph* (D-SAG). D-SAG is a graph where each field is represented by a node, and fields that are accessed together in time are connected by edges. Edges have weights, with higher weights signifying more contemporaneous accesses of a pair of fields. Figure 2 shows an abridged example of the LRU Handle data structure from RocksDB [3] and the corresponding D-SAG.

Our second contribution is a set of algorithms that analyze a D-SAG and recommend changes to data structures that improve spatial locality. These algorithms use graph clustering and produce recommendations to the programmer. Figure 3 shows the output of our algorithm for the RocksDB example in Figure 2: the recommendation is to collocate specific fields (found to be frequently accessed together) in the same class. Applying these changes to RocksDB reduces the runtime of a workload that stresses its memory table by 20%.

As a proof of concept, we developed a toolchain that automatically generates D-SAG from C and C++ code (using static and dynamic analysis) and recommends changes to data structures or classes. Our toolchain is based on the DINAMITE [19] LLVM pass,

```
struct LRUHandle {
  void* value;
  void (*deleter)(const ...);
  LRUHandle* next_hash;
  LRUHandle* next;
  LRUHandle* prev;
  size_t charge;
  size_t key_length;
  uint32_t refs;
  char flags;
  uint32_t hash;
  char key_data[1];
  // ...
};
```

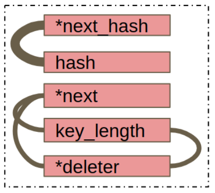


Figure 2: A snippet of a data structure in RocksDB and the corresponding D-SAG. Thicker edges represent more frequent co-temporal accesses.

```
### Pointer fields inlined:
  None


### New classes:

new class 0 {                // size: 12, MR: 0.55%, MRP: 72.96%
    *LRUHandle.next_hash;    // size: 8, MR: 0.18%, MRP: 23.27%
    LRUHandle.hash;          // size: 4, MR: 0.38%, MRP: 49.69%
}


new class 1 {                // size: 24, MR: 0.18%, MRP: 24.10%
    *LRUHandle.next;         // size: 8, MR: 0.10%, MRP: 12.61%
    LRUHandle.key_length;    // size: 8, MR: 0.06%, MRP: 7.66%
    *LRUHandle.deleter;      // size: 8, MR: 0.03%, MRP: 3.83%
}
```

Figure 3: Recommended data structure definitions produced by our toolchain for the code in Figure 2. MR is the LLC miss ratio, while MRP is the percentage contributed by this field to the overall miss ratio.

and works for C/C++ programs that can be compiled with LLVM 3.5. The analysis is primarily useful for memory-bound programs that use complex classes or data structures (as opposed to arrays of primitives, which are usually already optimized spatial locality). Based on these criteria, we were able to analyze ten memory-bound benchmarks from SPEC CPU2017, PARSEC, and RocksDB's db_bench, and improve performance by an average of 11% (geomean) and reduced cache misses by an average of 28% (geomean) for seven of these workloads.

The rest of the paper is organized as follows: Section 2 describes related work. Section 3 presents the D-SAG abstraction and the algorithms for its analysis. Section 4 describes the implementation. Section 5 presents the evaluation. Section 6 discusses limitations and future work. Section 7 concludes.

2 RELATED WORK

Prior studies that automatically identified DSS optimizations are summarized in Table 1. The focus of these studies was to identify one or two specific optimizations, while our goal is to design a common abstraction for many DSS techniques.

Table 1: Data Structure Splicing Optimizations Comparison

Optimization	Prior works	Our work
Class splitting	Yes [7] [13] [16] [24] [23]	Yes
Pointer-field inlining	Yes [8]	Yes
Fields reordering	Yes [7] [16] [10]	Yes
Class merging	No	Yes
Fields migrating	No	Yes

In "Cache-Conscious Structure Definition" [7], Chilimbi et al. focus on two optimizations: class splitting and field reordering. For class splitting, they use the abstraction of hot/cold fields: frequently accessed 'hot' fields are isolated in a separate class from infrequently accessed 'cold' fields. This abstraction does not enable DSS optimizations other than class splitting and may not improve cache line utilization if the co-located hot fields are not accessed contemporaneously.

Chilimbi's field-reordering optimization does rely on computing *affinity* between fields, which is very similar to our definition of affinity (see Section 3.2). However, their analysis is done on fields within a single class, so cross-class optimizations are not possible. In contrast, our D-SAG abstraction detects field affinities within a class as well as across classes. As a result, D-SAG is poised to detect *class merging* and *field migration* opportunities, whereas previously used abstractions were not (see Table 1).

Hundt et al. [13] modify a compiler to perform structure splitting and peeling by finding dead fields. Lack of runtime information may prevent idenitfying fields that are accessed together and those that generate many cache misses. That aside, Hundt's optimizations were performed on individual data structures only.

Lin et al. proposed a compiler framework that integrates data structure splitting and field reordering with array flattening [16]. For class splitting, they use a similar abstraction as Chilimbi's cache-conscious data structures: they split the class with fields whose access count is higher than the average. For field reordering, they proposed a simple heuristic to sort the fields according to their access count so that hot fields are grouped. (A similar strategy was used by Eimouri et al. [10].) The limitations of their class splitting approach is similar to that of Chilimbi's work. The limitations of their field reordering strategy are that cross-class optimizations are not possible, and grouping fields without considering their affinity may yield poor performance.

Dolby et al. use pointer field inlining to solve performance issues caused by software module isolation [8]. They analyze memory traces to inline pointer fields of objects that have "parent-child" or "one-to-one pointer field" relationship. This approach is effective, but does not produce optimizations besides pointer field inlining.

Zhong et al. [24] analyze "reference affinity" with LRU stack distance between objects, which is similar to our definition of affinity. They focus on trying out different thresholds of distance to identify opportunities for array regrouping and structure splitting. The analysis does not cover fields reordering or class merging in general.

Zhao et al. [23] also examine structure splitting, or array splitting in particular. They try different strategies for splitting, i.e., maximum splitting or splitting each field, frequency-based splitting, affinity-based splitting. Their approach does not cover optimizations other than class/structure splitting.

Our D-SAG abstraction is inspired by Miucin's access graphs [20]. In Miucin's access graphs, nodes represented individual *memory addresses*, and the graphs themselves were used to guide dynamic allocation of data structure instances. In D-SAG, however, nodes represent *data structure fields*, and the graphs are used to reorganize data structures in the application's source code.

3 ABSTRACTION AND ALGORITHMS

3.1 Requirements for a Common Abstraction

To understand the requirements for an effective common abstraction for DSS optimizations, let us consider two examples: the RocksDB code fragment from Figure 2 and the *canneal* code fragment in Figure 4.

The RocksDB code fragment (Figure 2) shows a data structure that accounts for most of the cache misses (~85%) in the RocksDB workload that stresses its memory table (see Section 5). The graph in the figure shows that the fields `next_hash` and `hash` are accessed both *frequently* (= both are included in the graph) and *together in time* (= they are connected by a thick edge). However, in the memory layout that corresponds to this definition, they are separated by six other fields, which are not accessed contemporaneously. Worse yet, they are likely to span across more than one cache line: for example, even if `LRUHandle` were allocated on a 64-byte cache line boundary, `next_hash` would be in the first line and `hash` would span the first and second lines. Similarly, fields `next`, `key_length`, and `deleter` are accessed together but separated by other fields; while they span only 48 bytes, `LRUHandle` allocation in the RocksDB code does not respect cache line boundaries, so in reality the three fields are likely to be allocated across two cache lines. To improve cache line usage, therefore, we would want to place `next_hash` and `hash`, as well as, separately, `next`, `key_length`, and `deleter`, close in memory. Observe that reasoning about this example requires knowing (a) which of a structure's fields are accessed often, and (b) which fields are accessed together.

The data structures in Figure 4 (from *canneal* in PARSEC [5], where they account for ~65% of all cache misses) demonstrate that this is not quite enough. The original code frequently accesses fields `fanin`, `fanout`, and `present_loc` from class `netlist_elem` *together in time*, while the field `item_name` is accessed infrequently. Moreover, accesses to `present_loc` dereference it and access its elements `x` and `y`. To improve cache line usage, then, we would want to separate `item_name` from `fanin`, `fanout`, inline the `x` and `y` subfields of `present_loc`, and place all four close in the address space. To reach this conclusion, we needed not only the intra-structure access frequencies as in the RocksDB example above, but also the ability to track these across different related structures.

In other words, a common abstraction must (a) identify frequently and infrequently accessed fields, and (b) capture contemporaneous accesses, both within and across data structure boundaries.

As the basis for this abstraction, we first define the term *access affinity*, to which we have been loosely referring as accesses occurring "together in time".

```
class location_t {
public:
  int x;
  int y;
};

class netlist_elem {
  string item_name;
  vector<netlist_elem*> fanin;
  vector<netlist_elem*> fanout;
  location_t* present_loc;
};
```

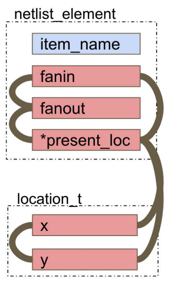


Figure 4: A snippet of a data structure in PARSEC canneal and the corresponding D-SAG.

```
### Pointer fields inlined:
  *netlist_elem.present_loc;

### New classes:

new class 0 {                // size: 56, MR: 0.61%, MRP: 22.47%
    location_t.x;            // size: 4, MR: 0.05%, MRP: 1.94%
    location_t.y;            // size: 4, MR: 0%, MRP: 0%
    netlist_elem.fanin;      // size: 24, MR: 0.31%, MRP: 11.40%
    netlist_elem.fanout;     // size: 24, MR: 0.25%, MRP: 9.13%
}

new class 1 {                // size 32, MR: 0.001%, MRP: 0.03%
    netlist_elem.item_name; // size 32, MR: 0.001%, MRP: 0.03%
}
```

Figure 5: Recommended data structure definitions produced by our tool for the code in Figure 4.

3.2 Access Affinity

To capture which fields are accessed together — and how closely together in time — we leverage Mattson's notion of *stack distance* [17]. Given accesses to two fields u and v in a memory access trace, stack distance is the number of accesses to unique data elements between the accesses to u and v; for example, in a trace *'uababv'*, the stack distance between u and v is two. Intuitively, the lower the stack distance, the more closely in time two fields are accessed.

Next, we observe that, for the purposes of optimizing spatial locality, only short stack distances matter. This is because two fields with a long stack distance may not result in an improved cache hit rate even if they are placed next to each other, as the cache line may well have been evicted between the two accesses.

We therefore define an *affinity event* as an occurrence, in a memory access trace, of a stack distance below a threshold t. In other words, if a pair of fields u and v were accessed with fewer than t other unique elements accessed in between, we detect an affinity event. We then define *access affinity* between a pair of fields as the number of affinity events between the two fields in the memory trace. (We discuss threshold selection in Section 4.3.) The pair of fields do not have to belong to the same object.

The concept of access affinity allows us to reason about a *pair* of fields; in the next section, we combine access affinity information for all data structures in the program in one abstraction that allows us to co-locate data that are frequently accessed together.

3.3 D-SAG

To reason about the relationships among different fields from different data structures, we construct the **D**ata **S**tructure **A**ccess **G**raph (D-SAG).

A D-SAG is as an undirected graph where each node represents a field in a data structure or a class. Each node includes a counter that indicates how many times the corresponding field was accessed. Edges carry weights, equal to the access affinities between the relevant pairs of fields. For example, an edge $u - v$ with a weight of 20 indicates that fields u and v were accessed within a threshold stack distance on twenty separate occasions. Because access counts are in general input-dependent, a D-SAG is specific to both the application (data structure definitions) and the memory access trace (access affinities).[1]

Figure 7 shows the D-SAG constructed from the memory access trace of the example code in Figure 6. (Such graphs are produced automatically by our tools.) Thicker edges represent stronger affinity.

```
class Bar {
public:
    int bar_a;
    int bar_b;
    int bar_c;
};

class Foo {
public:
    int foo_head;
    char foo_mid[64];
    Bar* foo_bar_p;
    int foo_tail;
};
```

```
class Large {
public:
    char large_a[64];
    char large_b[64];
    char large_c[64];
    char large_d[64];
    char large_e[64];
};
```

```
for (...) {
    out += foos[i].foo_head ^ foos[i].foo_tail;
    ...
    Bar* p = foos[i].foo_bar_p;
    out += p->bar_a ^ p->bar_b ^ p->bar_c;
    ...
    if (foos[i].foo_bar_p ...)
        out += strcmp(foos[i].mid, ...);
}

for (...) {
    ...
    out += Access(larges[i].large_a,
                  larges[i].large_c,
                  larges[i].large_e);
    ...
}
```

Figure 6: The data structure and the code accessing it for the D-SAG in Figure 7.

[1]We explain how we obtain memory access traces in Section 4.

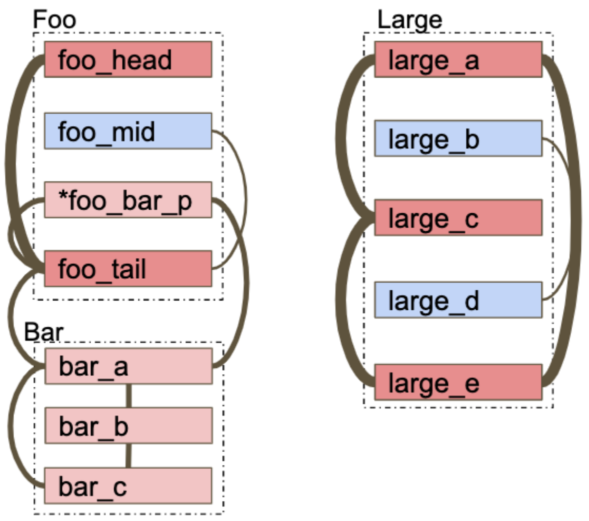


Figure 7: The D-SAG (G_0) for the code in Fig 6 under an example workload. Nodes represent data structure fields, with the colour representing access frequency (red = "hot", blue = "cold"). Edge weights represent access affinity between a pair of fields: the thicker the edge, the more often the fields are accessed together. Dashed outlines represent identify the data structure to which the outlined fields belong.

3.4 D-SAG Analysis

To demonstrate the usefulness of the D-SAG abstraction for optimizing spatial locality, we demonstrate how it can be used to express three different optimizations that restructure class fields to optimize for spatial locality: class splitting and merging, field inlining, and field reordering. We organize the optimizations as a three-stage pipeline to show how they can be applied together using the common D-SAG abstraction.

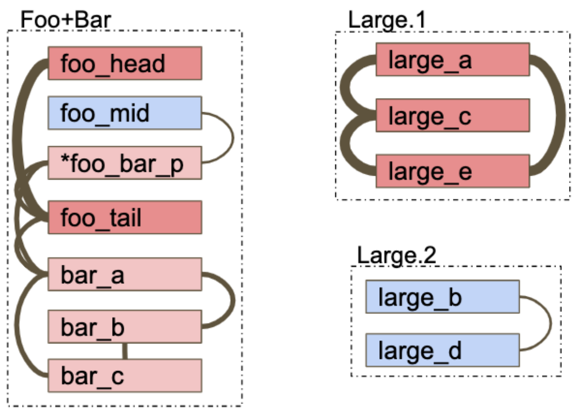


Figure 8: G_1, Stage 1, transformed from G_0 in Figure 7. This graph is manually crafted to help explain.

3.4.1 Stage 1: Class Splitting and Merging. Splitting and merging transforms a set of classes into new classes where (a) fields frequently accessed together are merged in the same class, and (b) fields that are rarely accessed together are split into different classes. The first aspect means that accessing one field will also bring into the cache fields that are likely to be accessed soon (i.e., spatial locality is improved); the second aspect means that fields that are *not* likely to be accessed soon are not brought into the cache (i.e., cache wastage is reduced).

To effect the splitting/merging optimization, we want to transform the D-SAG into several clusters, where each cluster consists of nodes that are connected by heavy edges. This is similar to the community detection problem, where the goal is to cluster the graph in such a way that the whole graph would have the best modularity score [21]. A high modularity score implies dense connections between the nodes within clusters but sparse connections between nodes in different clusters, which corresponds to our goal of optimizing spatial locality by placing frequently co-accessed data close by but rarely co-accessed data far away.

In our implementation, we use the multi-level community detection graph-clustering algorithm [6] to perform clustering: we empirically observed that this algorithm produces a high-quality clustering with little tuning (e.g., we do not need to specify the number of clusters) and good performance (faster than other candidates, e.g., k-means, spectral clustering, etc.).

Formally, this stage takes an input D-SAG $G_0(V, E, C_0)$, where

- V is the set of the nodes in the D-SAG,
- E is the set of weighted edges in the D-SAG, and
- C_0 is the set of clusters where each cluster c_i represents a data structure in the original code,

and produces an output graph $G_1(V, E, C_0, C_1)$, where

- C_1 is a set of clusters produced by the graph-clustering algorithm, with each cluster c_j includes a set of fields that are frequently accessed together.

For example, this optimization transforms G_0 in Figure 7 is transformed into G_1 in Figure 8. The class `Large` is split into class `Large.1` and class `Large.2` because the fields `large_b` and `large_d` are cold and do not have affinity to other fields of the same class (`large_a`, `large_d`, and `large_e`). Class `Foo` and class `Bar` are merged because of strong affinity, as indicated by the heavy edges.

3.4.2 Stage 2: Field Inlining. The field inlining optimization merges substructure fields referenced via a pointer into the enclosing structure if they are frequently accessed together with other enclosing structure fields. (We saw an example of this in the *canneal* code snippet in Figure 4, where the `netlist_elem` class contained a pointer to a `location_t` subclass with fields `x` and `y`.) Field inlining improves cache hit rates because the inlined subfields are more likely to be in the same cache line than if they were in a separate structure (spatial locality). This optimization also has a secondary effect of improving instruction-level parallelism, as the subfield accesses are no longer dependent on the data read from the enclosing structure and can be issued to the cache/MSHRs even if the enclosing structure access is a cache miss.

To effect field inlining, we start with the output of Stage 1, that is the graph $G_1(V, E, C_0, C_1)$ where C_0 clusters fields according to the

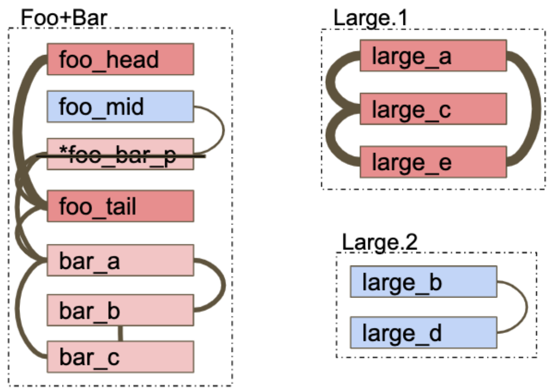


Figure 9: G_2, Stage 2, transformed from G_1 in Figure 8. This graph is manually crafted to help explain.

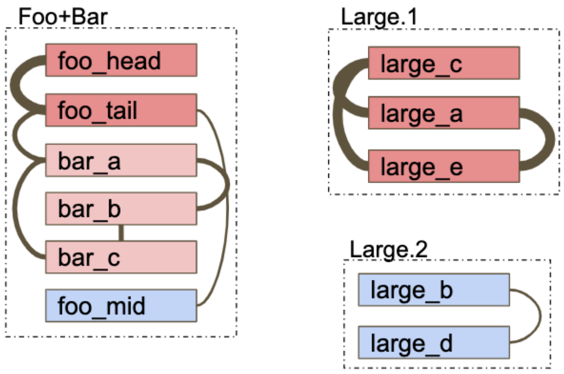


Figure 10: G_3, Stage 3, transformed from G_2 in Figure 9. This graph can be generated by our toolchain.

original class hierarchy and C_1 clusters fields according to access affinity. The C_1 clustering already brings together fields from the enclosing structure and the relevant fields from the substructure; all that remains, then, is to remove the substructure pointer if all fields were inlined.

$\forall v_j \in V$ and $\forall c_i \in C_1$, v_j is considered for removal if

- $v_j \in c_i$,
- v_j is a pointer type, and
- all fields of the original class pointed by v_j were merged into cluster c_i at Stage 1.

The inlining stage produces a graph $G_2(V, E, C_0, C_2)$ where

- C_2 is the clustering C_1 minus the pointer fields to classes with all fields inlined.

For example, the algorithm at this stage transforms G_1 in Figure 8 into G_2 in Figure 9: `foo_bar_p` is removed because it is of pointer type `Bar*` and the class `Bar` was fully merged into class `Foo` in Stage 1.

3.4.3 Stage 3: Field Reordering. Although Stage 1 ensures that a group of fields with high affinity is grouped together in the same data structure, it does not necessarily result in good spatial locality, as the fields with the highest affinity may not be placed next to each other. This matters especially if allocation is done without regard to cache line boundaries (as is the case with the RocksDB data structure in Figure 2), as the structure might begin anywhere in the cache line and two fields may end up in different lines even if they are within, say, 64 bytes of each other.

To decrease the impact of such splits on the cache miss rate, we organize fields within each data structure to ensure that fields with the highest access affinities are immediately next to each other in the memory layout. This ensures that even if structures are split across cache lines at random points, fields that are accessed together most frequently are likely to end up in the same cache line.

Stage 3 begins with the output from Stage 2: a graph $G_2(V, E, C_0, C_2)$ where C_2 clusters fields by affinity and inlines high-affinity substructures. The objective is to produce a graph $G_3(V, E, C_0, C_3)$ where

- $\forall c_i \in C_3$, fields in c_i are ordered to place fields with high affinity close together.

This goal is similar to the problem known as the weighted minimum linear arrangement (MinLA), where given a weighted graph $G(V, E)$, $|V|=n$, one must find a one-to-one function φ: V $\longrightarrow$ {1, ..., n} that minimizes $\sum_{ij \in V} |\varphi(i) - \varphi(j)| * e_{ij}$. MinLA is known to be NP-hard [12], and does not scale to the large number of fields in modern applications.

To achieve the goal efficiently, we developed an order-preserving variant of hierarchical clustering. We repeatedly group pairs of nodes (fields) inside each cluster in a descending order of edge weights (affinities) until all nodes are grouped into one in each cluster. Every time a pair of nodes is grouped, this pair is merged and treated as one node, with the edge weights from the merged nodes to any outside node combined via addition. When merging two nodes, we preserve their relative order in the original data structure; the resulting merged node inherits the order of the top component node that appears earlier in the structure. If the pair of nodes come from different data structures in the original code, then the one with more accesses is ordered first in the merged structure, based on the intuition that more frequently used fields will gain more benefit from being brought in with other data. Finally, the merged node's access frequency is the sum of access frequencies of its sub nodes.

Figure 10 shows an example of field reordering. In cluster `Foo+Bar`, node `foo_head` and node `foo_tail` are considered first because they are connected by the heaviest edge. Nodes `foo_head` and `foo_tail` are merged into one node, with `foo_head` before `foo_tail` according to the original field order. Next, `bar_a` is placed after `foo_tail` because `foo_tail-bar_a`) is the second heaviest edge, and the access frequency of `bar_a` is less than that of the previously merged node `foo_head+foo_tail`. The process continues for the remaining nodes, with `bar_b` grouped after `bar_a`, and `bar_c` after `bar_b`. Finally, `foo_mid` is grouped after `bar_c` due to its original order being before the merged node that begins with `foo_head`.

Figure 11 shows the final output generated by our toolchain (Section 4) after running the three-analysis pipeline on the code from Figure 6.

```
### Pointer fields inlined:
  *Foo.foo_bar_p;


### New classes:

new class 0 {                // size: 192, MR: 1.52%, MRP: 58.36%
    Large.large_a;           // size: 64, MR: 0.51%, MRP: 19.45%
    Large.large_e;           // size: 64, MR: 0.51%, MRP: 19.45%
    Large.large_c;           // size: 64, MR: 0.51%, MRP: 19.45%
}


new class 1 {                // size: 92, MR: 0.94%, MRP: 36.16%
    Foo.foo_head;            // size: 4, MR: 0.09%, MRP: 3.56%
    Foo.foo_tail;            // size: 8, MR: 0.70%, MRP: 27.12%
    Bar.bar_a;               // size: 4, MR: 0.05%, MRP: 1.92%
    Bar.bar_b;               // size: 4, MR: 0.04%, MRP: 1.64%
    Bar.bar_c;               // size: 4, MR: 0.05%, MRP: 1.92%
    Foo.foo_mid;             // size: 68, MR: 0.00%, MRP: 0.00%
}


new class 2 {                // size: 64, MR: 0.00%, MRP: 0.00%
    Large.large_d;           // size: 64, MR: 0.00%, MRP: 0.00%
    Large.large_b;           // size: 64, MR: 0.00%, MRP: 0.00%
}
```

Figure 11: Data structure definition suggestions: final output after Stage 3. 'Size' is the size of the field or the class, MR is the LLC miss ratio generated by this field, MRP is the normalized miss ratio percentage – the fraction contributed by this field to the overall miss ratio.

The optimizations are generated in different stages, but they will not conflict with each other, because the later stages only refine the output of the earlier stages (e.g., stages 2 and 3 cannot re-form clusters);

4 IMPLEMENTATION

In this section we describe our toolchain that embodies the abstraction and algorithms we propose in this paper. The toolchain takes an original program as input and suggests DSS optimizations as output, just like in the examples in Figures 3 and 5.

The entire workflow is shown in Figure 12. Phase ❶ involves profiling the program with Linux *perf* [1] (a) to determine whether it is memory-bound and (b) to identify the functions responsible for most of the cache misses.

Phase ❷ extracts data structure definitions from the binary and instrumenting the binary to produce the memory access trace.

After running the instrumented binary and recording the memory trace, Phase ❸ builds the D-SAG, analyses it and produces data structure splicing sugggestions.

Phase ❹ simulates the proposed suggestions by reorganizing the original memory trace as if the data structures were modified according to the suggestions of Phase ❸. The re-arranged trace is fed through the Dinero cache simulator [9] to see if the proposed changes would bear fruit.

Next, we describe each phase in more detail.

4.1 Phase 1: Identifying memory-bound programs and functions

We profile the program with Linux *perf* [1] to determine if it is memory-bound and to identify the functions responsible for many

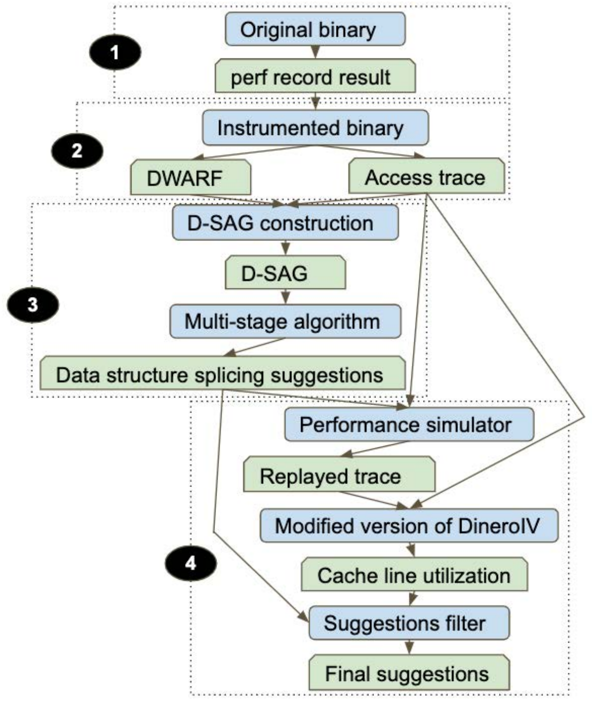


Figure 12: The pipeline of the proposed toolchain. The blue rectangles are the components of the pipeline, the green trapezoids are the outputs from the pointed components.

cache misses. Knowing the culprits allows us to instrument only those functions and not the entire program; as a result, the memory trace is shorter and the analysis phase completes faster.

We measure the L1, L2, and LLC misses and filter those programs whose L1 miss ratio below 3% and LLC miss ratio is below 1%. Functions with L1 miss ratio above 0.5% or LLC miss ratio above 0.2% are chosen for instrumentation.

4.2 Phase 2: Memory trace collection and static analysis

We obtain data structure definitions contained in the DWARF [18] binary. The program needs to be compiled with '-g' to make that information available. Data structure definitions will be used in Phase ❸, during D-SAG construction.

We compile the program with LLVM clang; during compilation, we also run the DINAMITE [19] instrumentation pass. DINAMITE instruments memory accesses in functions selected during Phase ❶ and the memory allocation routines. Combined with the data structure definitions, this information will allow us to determine when an object is allocated, what fields it contains, and when those fields are accessed[2].

[2]We infer the type of the object from the pointer casting instruction following an allocation.

4.3 Phase 3: D-SAG and Analysis

To construct the D-SAG, we parse the memory access trace. We set up a shadow heap to detect newly allocated objects, identify accesses to their fields and detect affinity events (Section 3.2).

For each entry in the memory access trace:

- if it is an allocation entry, we allocate the corresponding object(s) on the shadow heap and record its type;
- if it is a memory access to a dynamically allocated object, we find the corresponding object in the shadow heap and determine the field that was accessed by mapping the field's offset to the DWARF data structure definition.

If the access is not to a dynamically allocated object, we simply record it into the stash of recently accessed addresses, which we use to detect affinity events.

If the access is to a field in a dynamically allocated object, we create a new node in D-SAG, if this the first time we encounter an access to that field. We then examine the stash of recently accessed memory addresses to detect affinity events (fields accessed within the stack distance threshold from each other – see Section 3.2). If an affinity event between two fields is detected, we will either create an edge with the weight of one between the corresponding nodes, or increment the weight if the edge already exists.

We tried different stack distance thresholds for the applications we evaluated and found the value of ten to work best. It is small enough to not create too many edges and large enough to detect affinity between fields accessed contemporaneously. Tuning the threshold to the individual program and hardware was beyond the scope of this work.

We treat primitive type allocations (e.g., *int*, *double*, etc.) as a single-field class. For example, if an array of *int* is allocated at line 7 of a source file *A*, and another array of *int* is allocated at line 8 of same source file *A*, we will create two new classes: *class-A-l7* and *class-A-l8*. If the members of these arrays have a high affinity to each other, D-SAG will suggest merging the single-element "classes", which is essentially merging the two arrays so that their members are interleaved.

Upon processing the memory trace, we have the corresponding D-SAG, which we then analyze using the three-step process described in Section 3.4. The output of this phase are text files with DSS recommendations, like the ones shown in Figures 3 and 5.

4.4 Phase 4: Simulating proposed changes

To gauge the potential impact of the DSS optimizations recommended by Phase 3, we rearrange the original memory trace to reflect the new locations of fields after the recommended code changes. We feed the new trace through the DineroIV cache simulator [9], which we modified to also measure cache line utilization. As we show in Section 5, the miss rates produced by the simulator track those measured on real hardware when we apply the changes to the original programs. This means that we can use the output of the simulator to decide if the estimated reduction in the miss rates and the improvement in cache line utilization justify investing the effort into changing the actual code.

5 EVALUATION

5.1 Benchmarks

We evaluated our toolchain on C/C++ benchmarks from the SPEC CPU 2017 [4] and PARSEC [5] suites, as well as a modified version of the *read random* workload from RocksDB's db_bench [3] that stresses its memory table[3].

From the SPEC and PARSEC applications, we excluded: four as not compatible with LLVM-clang 3.5 (required by DINAMITE); three where heavy use of template types limited the static analysis the current version of our toolchain could handle; and one application where a custom memory allocator prevented our tools from tracking the allocated objects and collecting the traces. This is a limitation of our implementation rather than of our techniques — that is, a more sophisticated version of our tools could address these challenges. For the time being, however, we excluded these applications from our analysis.

This left us with 21 applications listed in Table 2. All of them were analyzed with our tools using the workflow shown in Figure 12. Eleven of these benchmarks had no opportunities for optimizations: they either already had near-perfect cacheline utilization (Group 2), a low cache miss rate (Group 3), or used a single array of primitive types or simple classes instead of complex classes or data structures (Group 4). The remaining applications in Group 1 went through the entire analysis pipeline; our tools suggested optimizations for all of them except *520.omnetpp_r*, *525.x264_r*, and *557.xz_r*.

Table 2: Benchmarks Categories

Categories	Benchmarks
Optimizable	
1. Sophisticated class and memory-bounded	rocksdb, canneal, streamcluster, fluidanimate, ferret, 505.mcf_r, 511.povray_r, 520.omnetpp_r, 557.xz_r, 525.x264_r
No optimization opportunities	
2. Cacheline usage near 100%	dedup, bodytrack, blacksholes, freqmine, swaptions, 508.namd_r
3. Not memory-bounded	544.nab_r, 541.leela_r, 500.perlbench_r
4. Simple Class Definition	519.lbm_r, 531.deepsjeng_r

5.2 Methodology

We evaluated the benchmarks on a machine with an Intel® CoreTM i5-7600K four-core CPU, with 32KB L1 instruction and 32KB L1 data caches per core, a 256KB L2 cache per core, and a unified 6MB LLC. The size of the cache line is 64 bytes.

We ran our tool pipeline on the benchmarks in Group 1 and manually applied the data structure changes suggested by our tools.

[3]We isolate the function that accesses the memory table, which accounts for the highest fraction of CPU utilization and LLC misses: 15% and 20%, respectively; we optimize/measure only that portion of the benchmark.

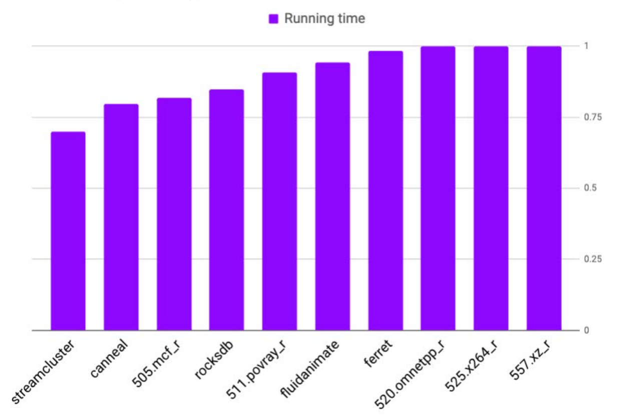


Figure 13: Runtime of the optimized version normalized to the original.

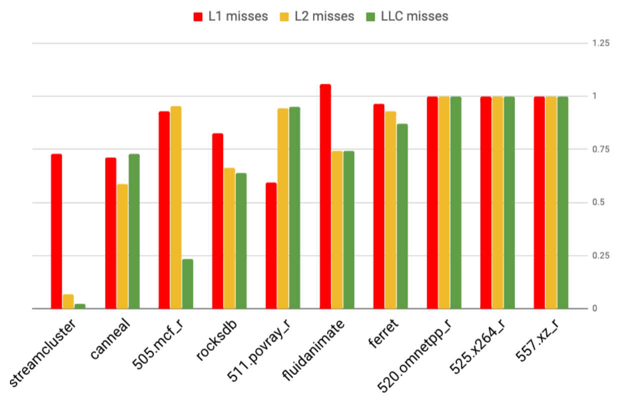
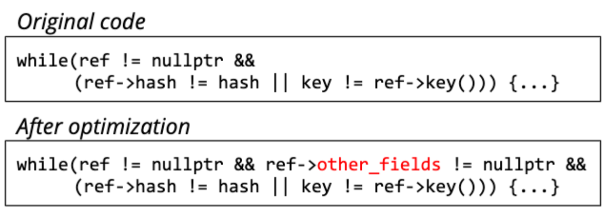


Figure 14: Cache misses of the optimized version normalized to the original.

E.g., if the tool suggests merging classes that make up multiple arrays, we replace these arrays with one array whose member is the merged-class object in the code; if the tool suggests splitting a class, we split all the pointers that point to this class. If the tool suggests a large number of changes, we only pick the ones that affect hot data structures (data structures account for at least 2% of LLC misses). In two cases, *505.mcf_r* and *511.povray_r*, the reorganization of data structures reverberated with the overwhelming number of changes that had to be made across the code (over 100 code locations in each benchmark). To reduce the manual effort, we isolated from these benchmarks only the functions that dominated the runtime (<90% CPU time).

5.3 Performance

We report the runtime and the caches miss rate (measured using *perf*) before and after optimizations. We also report the cache line utilization and the cache misses measured via simulation (as part of the pipeline in Figure 12).



Original code

```
while(ref != nullptr &&
     (ref->hash != hash || key != ref->key())) {...}
```

After optimization

```
while(ref != nullptr && ref->other_fields != nullptr &&
     (ref->hash != hash || key != ref->key())) {...}
```

Figure 15: Relevant RocksDB code before and after optimizations.

Figure 13 shows the runtime normalized to the original and Figure 14 shows the cache misses. The runtime improved for seven out of ten benchmarks, by up to 30% and by 11% on average (geometric mean). The three benchmarks that did not improve are the ones where our tools did not produce optimization suggestions. The reason is that the data structures were already organized according to access affinity.

For those benchmarks that did improve, the runtime improvement can be explained by the reduction in cache misses. For *fluidanimate* the number of L1 misses slightly increased, but the reduction in L2 and LLC misses provided a hefty compensation. Since the cost of an L1 miss (that hits in the L2 cache) is on the order of ten CPU cycles, while the cost of an L2 miss (that hits in the LLC) is on the order of 40 [2], this trade-off has a net positive effect on performance.

The number of executed instructions (not shown) remained unchanged in all applications except for RocksDB, where it increased by 4% when we applied the optimizations. That is because the logic of accessing the LRU table has changed (Figure 15 shows the original and the modified code). This price was worth paying, since the resulting improvements in spatial locality reduced the runtime by 20%.

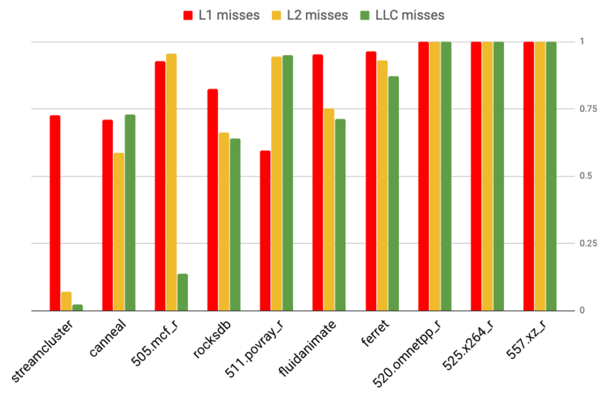


Figure 16: Simulated cache misses measured on rearranged traces, normalized to those measured on the original trace.

To get a further glimpse into the root cause of the observed performance improvements, we used an enhanced version of the DineroIV [9] cache simulator. We re-arranged the memory addresses in the trace according to the suggestions produced by our

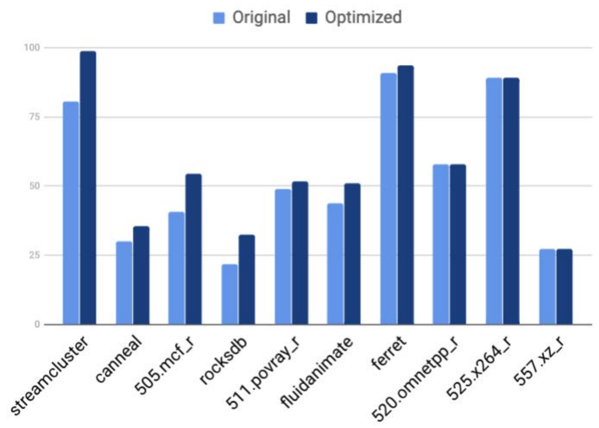


Figure 17: Cache line utilization for the original trace and the trace re-arranged accoring to optimization suggestions.

tools, isolating the effects of the improved memory layout from any secondary effects that the code changes might produce. Figure 16 shows the simulated miss rates obtained on the rearranged trace. The simulated miss rates improve very similarly to those measured on real hardware, suggesting that the effects we observed can be explained by a better memory layout.

Figure 17 reveals the driving force for these improvements. The new memory layout *increased the utilization of cache lines* for all benchmarks that experienced improved performance.

Data structure splicing is particularly effective for applications that allocate a large number of class objects consecutively (e.g., a large array). This is the case for *canneal*, *streamcluster*, and *505.mcf_r*, where our optimizations make the memory layout more compact. To further illustrate this effect, we study how our optimizations improved the performance of *canneal*.

5.4 Case Study - canneal

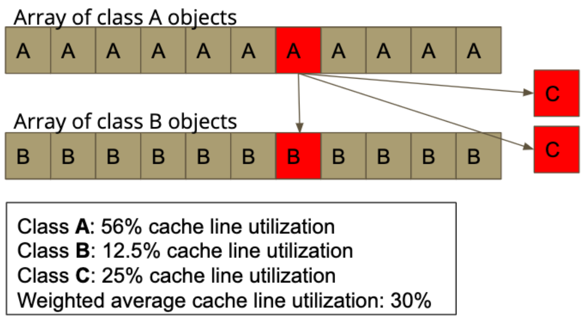

Array of class A objects
A A A A A A A A A A A
C
C
Array of class B objects
B B B B B B B B B B B
Class A: 56% cache line utilization
Class B: 12.5% cache line utilization
Class C: 25% cache line utilization
Weighted average cache line utilization: 30%

Figure 18: Access pattern of canneal before optimization

Figure 18 shows the original data access pattern in *canneal*. It repeatedly and randomly

- picks an element from a large array of class A, and
- reads the index and pointer fields in the class A object to access a class B object and two class C objects.

Essentially, there are four random memory accesses in each iteration: one to a class A object, one to a class B object, and one each to two class C objects. The combined footprint of all the objects of classes A, B, and C is much larger that the LLC, which means that the four random accesses result in four cache misses most of the time. The cache line utilization of objects in class A, B, C are 56%, 12.5% and 25% respectively, which gives an average cache line utilization of 30%.

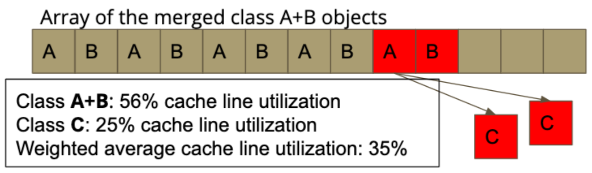

Array of the merged class A+B objects
A B A B A B A B A B
Class A+B: 56% cache line utilization
Class C: 25% cache line utilization
Weighted average cache line utilization: 35%
C
C

Figure 19: Access pattern of canneal after optimizations.

Since the class B object is always accessed through the index field of a class A object, our toolchain suggests to move the fields in class B to class A, which is shown in Figure 5. Figure 19 shows the access pattern after applying the optimizations. The new access pattern gives an average cache line utilization of 36%.

Because of the merge between classes A and B, in each iteration the number of random memory accesses drops from four to three, making it possible in theory to reduce the number of cache misses by 25%, and, indeed, the actual reduction in the cache miss rate and the runtime is very close to 25%.

6 DISCUSSION AND FUTURE WORK

6.1 Automating recommendations

Our tools *recommend* optimizations to a developer as opposed to applying them to the source code automatically. This has both advantages and limitations. The advantage is that the bar for tool adoption is low: a developer does not have to commit to an experimental compiler, and the resulting code readability (and ease of debugging) will not suffer, because the changes are applied by the developer. The downside is that sometimes the amount of manual changes may be overwhelming (as was the case for two applications in our study). On the other hand, automatic source code transformations are not always easy to apply, because some optimizations (e.g. splitting inherited classes in C++) are hard to apply without human decision. Understanding how to best combine the simplicity of hand-applied changes and the efficiency of automatic transformations is a possible avenue for future work.

When doing automatic transformations, we need to consider memory safety. For instance, when migrating a field from one class to another, which essentially increases the size of the class, we want to make sure that the behavior is still correct and does not raise issues like buffer overflow. Additionally, field migration or reordering may cause problems, because in languages like C/C++ an application may access the field via its offset, using pointer arithmetic. Analyzing the access pattern for safety using the memory access trace is not sufficient, because the access pattern may change if the input changes.

Since C/C++ are not memory-safe, we do not believe that recommendations produced by our tools can be applied automatically in the general case. We believe that a semi-automation tool, like the one we proposed, is a more practical approach.

6.2 Algorithm Validity and Correctness

It is known that the problem of optimizing the cache hit rate (and hence the problem of data structure layout) is NP-hard [12, 15]. Our algorithm uses a greedy, heuristic approach. While we cannot prove or claim optimality, the advantage is simplicity and speed of the algorithm (D-SAG construction and analysis took between 30 minutes and an hour on our benchmarks).

In the current implementation, we run two kinds of graph analysis on the same trace at once: graph clustering and inter-cluster analysis. We would like to understand whether repeating all or some these analyses several times after the data structures were reorganized according to the eariler output of the algorithm would yield better results.

6.3 Threats to Effectiveness

We used a pointer-type inference method to decide the type of an allocated object. Though this works for most C and C++ programs, generic pointer casting can still invalidate this mechanism, e.g., an allocated object that referred to by a pointer that is later cast to another type might be miss-labeled in our approach. A more sophisticated type inference method (e.g. combining static analysis with dynamic information) could mitigate this issue.

Merging classes with significantly different numbers of allocated objects sometimes might not make sense, e.g., the D-SAG might detect high affinity between the field of a singleton class and the field of class that is member of a large array. We can address this issue by adding a constraint in Stage 1 where edges between fields of different classes would be removed if the classes greatly differ in the number of allocated objects.

7 CONCLUSION

In this paper, we present a novel method to produce diversified types of data structure splicing optimizations under one single abstraction. We introduce the D-SAG (**D**ata **S**tructure **A**ccess **G**raph), which can reveal the access pattern and access affinity among fields of data structures. Based on D-SAG, we present a multi-stage algorithm that can analyze the graph and produce comprehensive types of data structure splicing optimizations suggestions. We measure the preliminary version of our D-SAG-based toolchain on a diverse set of workloads. On applications that are amenable to our optimizations (i.e., do not already have full cache line utilization), our technique reduces cache misses by an average of 28% (geomean) and improves performance by an average of 11% (geomean).

REFERENCES

[1] 2019. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/

[2] 2019. Performance Analysis Guide for Intel® CoreTM i7 Processor and Intel® XeonTM 5500 processors. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

[3] 2019. RocksDB | A persistent key-value store. https://rocksdb.org/

[4] 2019. Standard Performance Evaluation Corporation. https://www.spec.org/

[5] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton University.

[6] Vincent 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 (04 2008). https://doi.org/10.1088/1742-5468/2008/10/P10008

[7] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. 1999. Cache-conscious Structure Definition. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99).* ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/301618.301635

[8] Julian Dolby and Andrew Chien. 2000. An Automatic Object Inlining Optimization and Its Evaluation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00).* ACM, New York, NY, USA, 345–357. https://doi.org/10.1145/349299.349344

[9] Jan Edler and Mark D. Hill. 1998. Dinero IV: trace-driven uniprocessor cache simulator.

[10] Taees Eimouri, Kenneth B. Kent, Aleksandar Micic, and Karl Taylor. 2016. Using Field Access Frequency to Optimize Layout of Objects in the JVM. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16).* ACM, New York, NY, USA, 1815–1818. https://doi.org/10.1145/2851613.2851942

[11] 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 *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII).* ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2150976.2150982

[12] Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA.

[13] Robert Hundt, Sandya Mannarswamy, and Dhruva Chakrabarti. 2006. Practical Structure Layout Optimization and Advice. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06).* IEEE Computer Society, Washington, DC, USA, 233–244. https://doi.org/10.1109/CGO.2006.29

[14] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon. 2012. Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierarchy. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture.* 376–388. https://doi.org/10.1109/MICRO.2012.42

[15] Rahman Lavaee. 2016. The Hardness of Data Packing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16).* ACM, New York, NY, USA, 232–242. https://doi.org/10.1145/2837614.2837669

[16] Jin Lin and Pen-Chung Yew. 2010. A Compiler Framework for General Memory Layout Optimizations Targeting Structures. In *Proceedings of the 2010 Workshop on Interaction Between Compilers and Computer Architecture (INTERACT-14).* ACM, New York, NY, USA, Article 5, 8 pages. https://doi.org/10.1145/1739025.1739033

[17] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. 1970. Evaluation Techniques for Storage Hierarchies. *IBM Syst. J.* 9, 2 (June 1970), 78–117. https://doi.org/10.1147/sj.92.0078

[18] Michael J. Eager. 2012. Introduction to the DWARF Debugging Format. http://www.dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf

[19] Svetozar Miucin, Conor Brady, and Alexandra Fedorova. 2016. End-to-end Memory Behavior Profiling with DINAMITE. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016).* ACM, New York, NY, USA, 1042–1046. https://doi.org/10.1145/2950290.2983941

[20] Svetozar Miucin and Alexandra Fedorova. 2018. Data-driven Spatial Locality *(Memsys 2018).* ACM, New York, NY, USA.

[21] M. E. J. Newman. 2006. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences* 103, 23 (2006), 8577–8582. https://doi.org/10.1073/pnas.0601602103 arXiv:https://www.pnas.org/content/103/23/8577.full.pdf

[22] Moinuddin K Qureshi, M Aater Suleman, and Yale N Patt. 2007. Line distillation: Increasing cache capacity by filtering unused words in cache lines. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA).*

[23] Peng Zhao, Shimin Cui, Yaoqing Gao, Raúl Silvera, and José Amaral. 2005. Forma : A framework for safe automatic array reshaping. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30 (01 2005), 2. https://doi.org/10.1145/1290520.1290522

[24] Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. 2004. Array Regrouping and Structure Splitting Using Whole-program Reference Affinity. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04).* ACM, New York, NY, USA, 255–266. https://doi.org/10.1145/996841.996872