Data Structure Splicing

by

Lyuyu Ye

B.A.Sc, Simon Fraser University, 2016
B.Eng, Zhejiang University, 2016

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

June 2019

© Lyuyu Ye, 2019
The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

**Data Structure Splicing**

submitted by **Lyuyu Ye** in partial fulfillment of the requirements for the degree of **Master of Applied Science in Electrical and Computer Engineering**.

**Examine Committee:**

Alexandra Fedorova, Electrical and Computer Engineering  
*Supervisor*

Mieszko Lis, Electrical and Computer Engineering  
*Committee Member, Co-reader*

Sathish Gopalakrishnan, Electrical and Computer Engineering  
*Committee Member, Chair*
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 previously proposed 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.
Lay Summary

Data structures code is usually written unintentionally poorly for access locality, especially for modern sophisticated software. However, hardware requires locality to perform efficiently. It is hard to reason about locality without an explicit guide or insight in runtime workload. We propose D-SAG (Data Structure Access Graph) and the corresponding algorithm to solve the issue. Our solution automatically analyzes the runtime profile and provides developers with various types of code change suggestions regarding optimizing data structures definitions. The proposed solution has the potential to improve access locality, thus improves cache usages and performance for memory-bounded applications.
Preface

This thesis is an extension of an in-submission paper of the Memsys 19 conference filed as

- L. Ye, M. Lis, and A. Fedorova. “A Unifying Abstraction for Data Structure Splicing”.

I was the lead investigator on the work. All of the research was on my own. I did the writing with editorial help and technical advice from co-authors.
# Table of Contents

Abstract .......................................................... iii

Lay Summary ..................................................... iv

Preface ............................................................. v

Table of Contents ................................................. vi

List of Tables ..................................................... viii

List of Figures ................................................... ix

Acknowledgments ................................................ xi

1 Introduction .................................................... 1

2 Related Work .................................................. 6

3 Abstraction and Algorithms ................................. 8
   3.1 Requirements for a Common Abstraction ............... 8
   3.2 Access Affinity ........................................... 10
   3.3 D-SAG ..................................................... 11
   3.4 D-SAG Analysis ........................................... 11
       3.4.1 Stage 1: Class Splitting and Merging ........... 11
       3.4.2 Stage 2: Field Inlining ............................ 15
       3.4.3 Stage 3: Field Reordering ......................... 16
List of Tables

Table 2.1  Data Structure Splicing Optimizations Comparison . . . . . . . 6
Table 5.1  Benchmarks Categories . . . . . . . . . . . . . . . . . . . . . 26
List of Figures

Figure 1.1 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 \cite{10} cache simulator. In most applications, very little of a cache line is used before the line is evicted. \hspace{1cm} 2

Figure 1.2 A snippet of a data structure in RocksDB and the corresponding D-SAG. Thicker edges represent more frequent co-temporal accesses. \hspace{1cm} 5

Figure 1.3 Recommended data structure definitions produced by our toolchain for the code in Figure 1.2. MR is the LLC miss ratio, while MRP is the percentage contributed by this field to the overall miss ratio. \hspace{1cm} 5

Figure 3.1 A snippet of a data structure in PARSEC canneal and the corresponding D-SAG. \hspace{1cm} 9

Figure 3.2 Recommended data structure definitions produced by our tool for the code in Figure 3.1. \hspace{1cm} 10

Figure 3.3 The data structure and the code accessing it for the D-SAG in Figure 3.4. \hspace{1cm} 12
Figure 3.4  The D-SAG (G₀) for the code in Fig 3.3 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. .................. [13]

Figure 3.5  G₁, Stage 1, transformed from G₀ in Figure 3.4. This graph is manually crafted to help explain. .................. [14]

Figure 3.6  G₂, Stage 2, transformed from G₁ in Figure 3.5. This graph is manually crafted to help explain. .................. [15]

Figure 3.7  G₃, Stage 3, transformed from G₂ in Figure 3.6. This graph can be generated by our toolchain. .................. [17]

Figure 3.8  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. .................. [19]

Figure 4.1  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. .................. [21]

Figure 5.1  Runtime of the optimized version normalized to the original. . . [27]
Figure 5.2  Cache misses of the optimized version normalized to the original. [28]
Figure 5.3  Relevant RocksDB code before and after optimizations. . . . [28]
Figure 5.4  Simulated cache misses measured on rearranged traces, normalized to those measured on the original trace. ............ [29]
Figure 5.5  Cache line utilization for the original trace and the trace rearranged according to optimization suggestions. ............ [30]
Figure 5.6  Access pattern of canneal before optimization ............ [31]
Figure 5.7  Access pattern of canneal after optimizations ............ [31]
Acknowledgments

I would like to express my deepest appreciation to my supervisor Prof. Alexandra Fedorova, whose guidance made this research possible. I own particular thanks to my graduated lab mate Dr. Svetozar Miucin, whose work provides the foundation of my research. I would also like to thank all of the contributors and co-authors for all their hard work. I offer my enduring gratitude to the faculty, staff and my fellow students at UBC, who have inspired me to continue my work in this field.

Special thanks go to my family for believing in me throughout this journey and my friends who were always there for me.

Finally, I would like to thank Google for providing me with internship opportunity that inspired and help me in my research direction.
Chapter 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 [15]. Figure 1.1 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 exceed the capacities of feasible caches [12], 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., 15] 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...
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 accessed fields are segregated from “cold” or infrequently accessed fields [7, 14, 17]; field reordering — when fields that are accessed together are co-located in the data structure [7, 11]; 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 op-
timizations. 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 pro-
duce 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 thesis, 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 1.2 shows an abridged
example of the LRU Handle data structure from RocksDB [3] and the correspond-
ing 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 al-
gorithms use graph clustering and produce recommendations to the programmer.
Figure 1.3 shows the output of our algorithm for the RocksDB example in Fig-
ure 1.2: 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 recom-
mends changes to data structures or classes. Our toolchain is based on the DI-
NAMITE [20] LLVM pass, 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 thesis is organized as follows: Chapter 2 describes related work. Chapter 3 presents the D-SAG abstraction and the algorithms for its analysis. Chapter 4 describes the implementation. Chapter 5 presents the evaluation. Chapter 6 discusses limitations and future work. Chapter 7 concludes.
Figure 1.2: A snippet of a data structure in RocksDB and the corresponding D-SAG. Thicker edges represent more frequent co-temporal accesses.

```c
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 1.3: Recommended data structure definitions produced by our toolchain for the code in Figure 1.2. MR is the LLC miss ratio, while MRP is the percentage contributed by this field to the overall miss ratio.

```c
### 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%
}
```
Chapter 2

Related Work

Prior studies that automatically identified DSS optimizations are summarized in Table 2.1. 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>
<thead>
<tr>
<th>Optimization</th>
<th>Prior works</th>
<th>Our work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class splitting</td>
<td>Yes [7] [14] [17]</td>
<td>Yes</td>
</tr>
<tr>
<td>Pointer-field inlining</td>
<td>Yes [8]</td>
<td>Yes</td>
</tr>
<tr>
<td>Fields reordering</td>
<td>Yes [7] [17] [11]</td>
<td>Yes</td>
</tr>
<tr>
<td>Class merging</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Fields migrating</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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 affini-
ties 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 2.1).

Hundt et al. [14] modify a compiler to perform structure splitting and peeling by finding dead fields. Lack of runtime information may prevent identifying 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 [17]. 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. [11].) 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.

Our D-SAG abstraction is inspired by Miucin’s access graphs [19]. 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.
Chapter 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 1.2 and the canneal code fragment in Figure 3.1.

The RocksDB code fragment (Figure 1.2) 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 3.1 (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”.

```java
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 3.1:** A snippet of a data structure in PARSEC canneal and the corresponding D-SAG.
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 [18]. 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 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 **Data Structure Access Graph (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 $u \rightarrow v$ 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 3.4 shows the D-SAG constructed from the memory access trace of the example code in Figure 3.3; (Such graphs are produced automatically by our tools.) Thicker edges represent stronger affinity.

### 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.

#### 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

---

\(^1\)We explain how we obtain memory access traces in Section 4.
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).

Figure 3.3: The data structure and the code accessing it for the D-SAG in Figure 3.4.
Figure 3.4: The D-SAG ($G_0$) for the code in Fig. 3.3 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.

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.
Figure 3.5: G₁, Stage 1, transformed from G₀ in Figure 3.4. This graph is manually crafted to help explain.

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 G₀(V, E, C₀), where

- V is the set of the nodes in the D-SAG,
- E is the set of weighted edges in the D-SAG, and
- C₀ is the set of clusters where each cluster cᵢ represents a data structure in the original code,

and produces an output graph G₁(V, E, C₀, C₁), where

- C₁ is a set of clusters produced by the graph-clustering algorithm, with each cluster cⱼ includes a set of fields that are frequently accessed together.
For example, this optimization transforms $G_0$ in Figure 3.4 into $G_1$ in Figure 3.5. 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

![Figure 3.6: $G_2$, Stage 2, transformed from $G_1$ in Figure 3.5. This graph is manually crafted to help explain.](image)

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 3.1 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 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.

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

- $v_j \in c_i$,
- $v_j$ 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 3.5 into $G_2$ in Figure 3.6: $\text{foo.bar.p}$ is removed because it is of pointer type $\text{Bar}^*$ and the class $\text{Bar}$ was fully merged into class $\text{Foo}$ 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 1.2), 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.

![Diagram](image)

**Figure 3.7:** G₃, Stage 3, transformed from G₂ in Figure 3.6. This graph can be generated by our toolchain.

Stage 3 begins with the output from Stage 2: a graph G₂(V, E, C₀, C₂) where C₂ clusters fields by affinity and inlines high-affinity substructures. The objective is to produce a graph G₃(V, E, C₀, C₃) where

- ∀ cᵢ ∈ C₃, fields in cᵢ 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 → {1, ..., n} that minimizes \( \sum_{i,j \in V} |φ(i) - φ(j)| * e_{ij} \). MinLA is known to be NP-hard [13], 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 3.7 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 3.8 shows the final output generated by our toolchain (Section 4) after running the three-analysis pipeline on the code from Figure 3.3.

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);
Figure 3.8: 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.
Chapter 4

Implementation

In this section we describe our toolchain that embodies the abstraction and algorithms we propose in this thesis. The toolchain takes an original program as input and suggests DSS optimizations as output, just like in the examples in Figures 1.3 and 3.2.

The entire workflow is shown in Figure 4.1. Phase 1 involves profiling the program with Linux perf[2] (a) to determine whether it is memory-bound and (b) to identify the functions responsible for most of the cache misses.

Phase 2 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 3 builds the D-SAG, analyses it and produces data structure splicing suggestions.

Phase 4 simulates the proposed suggestions by reorganizing the original memory trace as if the data structures were modified according to the suggestions of Phase 3. The re-arranged trace is fed through the Dinero cache simulator [10] 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* [2] to determine if it is memory-bound and to identify the functions responsible for many 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 [9] binary. The program needs to be compiled with `-g` to make that information available. Data structure definitions will be used in Phase 3 during D-SAG construction.

We compile the program with LLVM clang; during compilation, we also run the DINAMITE [20] instrumentation pass. DINAMITE instruments memory accesses in functions selected during Phase 1 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.

### 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.

---

1 We infer the type of the object from the pointer casting instruction following an allocation.
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 1.3 and 3.2.

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 [10], 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.
Chapter 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 LRU table. 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 5.1. All of them were analyzed with our tools using the workflow shown in Figure 4.1. Eleven of these benchmarks had no opportunities for optimizations: they either already had near-perfect cache-line 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.

---

1 We isolate the function that accesses the LRU 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.
(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 5.1: Benchmarks Categories

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

5.2 Methodology

We evaluated the benchmarks on a machine with an Intel® Core™ 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. 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
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 4.1).

Figure 5.1 shows the runtime normalized to the original and Figure 5.2 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 [1], 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 5.3 shows the original and the modified code). This price was worth paying, since the resulting improvements in spatial locality reduced the runtime by 20%.

\[ ... \]
To get a further glimpse into the root cause of the observed performance improvements, we used an enhanced version of the DineroIV [10] cache simulator. We re-arranged the memory addresses in the trace according to the suggestions produced by our tools, isolating the effects of the improved memory layout from any secondary effects that the code changes might produce. Figure 5.4 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 5.5 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 s05.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

Figure 5.6 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 than 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%.

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 3.2. Figure 5.7 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%.
Chapter 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 [13,16]. 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 earlier 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.
Chapter 7

Conclusion

In this thesis, we present a novel method to produce diversified types of data structure splicing optimizations under one single abstraction. We introduce the D-SAG (Data Structure Access Graph), 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).
Bibliography


36


Appendix A

Hardware

All experiments in Chapter 5 are performed on the following machine:

- Intel Core i5-7600K CPU @ 3.80GHz, 4 Cores, 32K * 4 L1d cache, 256K * 4 L2 cache, 8192K L3 cache.