Data-driven Spatial Locality

by

Svetozar Miucin

B.Sc., University of Novi Sad, 2010
M.Sc., University of Novi Sad, 2011

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

DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate and Postdoctoral Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

January 2019

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the dissertation entitled:

Data-driven Spatial Locality

submitted by **Svetozar Miucin** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy** in **Electrical and Computer Engineering**

Examining committee:

Alexandra Fedorova, Electrical and Computer Engineering

Supervisor

Ivan Beschastnikh, Computer Science

Supervisory Committee Member

Mieszko Lis, Electrical and Computer Engineering

Supervisory Committee Member

Michael J Feeley, Computer Science

University Examiner

Raymond Ng, Computer Science

University Examiner

Additional Supervisory Committee Members:

James Larus, École Polytechnique Fédérale de Lausanne

External Examiner

Abstract

Over the past decades, core speeds have been improving at a much higher rate than memory bandwidth. This has caused the performance bottlenecks in modern software to shift from computation to data transfers. Hardware caches were designed to mitigate this problem, based on the principles of temporal and spatial locality. However, with the increasingly irregular access patterns in software, locality is difficult to preserve. Researchers and practitioners devote a lot of time and effort to improving memory performance from the software side. This is done either by restructuring the code to make access patterns more regular, or by changing the layout of data in memory to better accommodate caching policies. Experts often use correlations between the access pattern of an algorithm and properties of the objects it operates on to devise new ways to lay data out in memory. Prior work has shown the memory layout design process to be largely manual and difficult enough to result in high level publications.

Our contribution is a set of tools, techniques and algorithms for automatic extraction of correlations between data and access patterns of programs. In order to collect a sufficient level of details about memory accesses, we present a compiler-based access instrumentation framework called DINAMITE. Further, we introduce access graphs, a novel way of representing spatial locality properties of programs which are generated from memory access traces. We use access graphs as a basis for Hierarchical Memory Layouts – a novel algorithm for estimating performance improvements to be gained from better data layouts. Finally, we present our Data-Driven Spatial Locality techniques which use the information available from previous steps to automatically extract the correlations between data and access patterns commonly used by experts to inform better layout design.

Lay Summary

Over the past decades, the disparity between processor and main memory speeds has grown significantly. Many important applications today suffer from poor memory performance. To improve these applications, experts manually tune the placement of data in memory. This process requires a deep understanding of the algorithm and the underlying hardware on which it runs. This work presents insights and analysis into how experts create performant memory layouts, and proposes new abstractions, algorithms and techniques to automate parts of the process. The proposed solutions have the potential of helping performance-minded engineers to improve data layout in their programs.

Preface

Chapter 2 is a modified version of our arxiv technical report filed as:

- S. Miucin, C. Brady, and A. Fedorova. "DINAMITE: A modern approach to memory performance profiling." arXiv preprint arXiv:1606.00396 (2016)., technical report

A shorter, peer-reviewed version of this work was published, ©2016 Association for Computing Machinery as:

- S. Miucin, C. Brady, and A. Fedorova. "End-to-end memory behavior profiling with DINAMITE." Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2016.

I was the lead investigator on the project. Conor Brady helped build the infrastructure to connect DineroIV [35] cache simulator and DINAMITE compiled binaries, as well as implemented and wrote the matter appearing in 2.3.2. I wrote the rest of the article and conducted all other experiments in the chapter. Other authors provided editorial and technical advice.

Chapter 3 is a modified version of previously published peer-reviewed work, ©2018 Association for Computing Machinery:

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

I was the lead investigator on the project. All of the research and writing is my own, with editorial and technical advice from co-authors.

The work in 3.5.4 was conducted after the "Data-driven Spatial Locality" publication. I wrote all the text within it and conducted all described experiments.

Table of Contents

List of Tables

List of Figures

List of Code Listings

Acknowledgements

I extend my sincerest thanks to my advisor Dr. Alexandra Fedorova, whose guidance made this research possible and made me the researcher I am today. I would also like to thank all of the contributors and co-authors for all their hard work.

I would like to thank my family for believing in me throughout this journey and my friends who were always there for me.

Finally, I would like to thank Mitacs, STMicroelectronics and Oracle Labs for providing me with internship and co-op opportunities that helped put some of my work into context.

Dedication

To my wife, Nevena, whose love, support and brilliant discussions on scientific methodology made life orders of magnitude more enjoyable.

Chapter 1

Introduction

Cache-based computer architectures are ubiquitous. They date back to 1970s, when hardware architects introduced them to tackle the growing divide between CPU speeds and RAM bandwidth [1] [21].

Caches are banks of memory that are orders of magnitude faster than DRAM, but also orders of magnitude smaller in size. Their purpose is to hold a portion of the working set close to the cores for quick access. Caches decide what data to keep local based on two assumptions about software behaviour: spatial and temporal locality [70].

Spatial locality means that if a program accesses one location in memory, it is likely to access its neighbouring locations in the near future. This principle is reflected in the way cache memories transfer data. The unit of transfer between cache memory and RAM is called a *cache line* and is typically 64B in size. Furthermore, most modern caches are equipped with *next line prefetchers* which take assumptions about spatial locality a step further and issue sequential memory requests in advance.

Temporal locality means that if a program accesses one location, it is likely to access it again in the near future. This is reflected in cache line replacement policies such as Least Recently Used (LRU). Due to the closed nature of cache implementations in hardware, we cannot be sure which replacement policies are implemented in production[1], however recent research [2] shows that temporal locality is still one of the main considerations in designing caches.

The entire mechanism of transferring data to and from DRAM is automatically executed in hardware. Because of this, cache-based systems are very easy to program compared to Explicitly Managed Memory systems described in the prologue. The user does not need to worry about which data are currently in the cache, they just need to ensure that their programs exhibit good spatial and temporal locality.

As programs become more complex, preserving spatial and temporal locality becomes difficult. Studies from as far back as two decades ago [7]

[1] reverse-engineering work exists [6] suggesting modified Pseudo-LRU

[2] such as Dynamic Re-reference Interval Prediction [39] and Protection Distances [39]

show that modern database software spends roughly 50% of its time stalled, waiting on memory transfers to complete. This trend continued into the era of cloud computing [30], and the poor memory performance is attributed, among other things, to the increase in working set size and irregular access patterns. Improving locality of access has been shown to increase the overall performance of programs, in some cases as much as an order of magnitude [45] [43] [61] [14] [32] [75]. However, optimizing memory performance is difficult. Significant advances have been made in understanding and optimizing various array access patterns – static analysis of loops [22] [24], loop tiling [57] [72], etc. Dynamically allocated pointer-based data structures, such as lists, graphs, meshes, tell a different story. Solutions are typically custom-tailored to a specific algorithm and data structure combination. One approach to optimizing memory performance of programs relies on **changing the layout of objects in memory** to achieve better spatial locality given the algorithm's access pattern. The intuition for making better memory layouts is to **put objects that are frequently accessed together close to each other in memory.** Prior work in the area [10] [42] [69] [65] [17] [25] [75] [67] [36] [37] [76] [50] led us to two observations:

1. Each new data structure and algorithm requires a significant effort in understanding the access patterns. This is done by experts in both the domain and memory optimization areas.

2. Often, layout solutions are guided by the data itself. For example, GraphChi [42] groups edges based on their destination node, and sorts such groups based on their source nodes, allowing iterative algorithms to make mostly linear accesses to storage when fetching data.

The rest of the work in this dissertation is based on the second observation *that the in-domain properties of objects can be used to inform better layouts*, and aims to reduce the amount of effort and expert knowledge noted in the first observation.

Chapter 2 presents our work in the extraction of memory access data from programs, along with tools to process the said data. Chapter 3 introduces a novel graph-based framework to capture the spatial locality properties of a program. It builds on this framework a set of techniques and tools aimed at characterizing the potential for performance improvement, and finally extracting the kind of expert knowledge identified in prior memory layout work. This expert knowledge entails **understanding the connections between the access pattern of the algorithm and properties**

of the accessed data objects, and how they can be used to derive better layouts.

Hypothesis 1.1 *The* primary hypothesis *of this dissertation is that the type of expert knowledge about the relationship of data and access patterns used in prior work can be extracted automatically from full memory trace accesses.*

Hypothesis 1.2 *The* secondary hypothesis *is that the extracted expert knowledge can be used directly in conjunction with hint-based allocators to improve memory performance of programs.*

The main contribution of this dissertation is the **validation of hypotheses 1.1 and 1.2**. Mainly, we have built a **semi-automated method for improving information about data and access pattern relationships from programs**. To that end, we contribute the following:

- A compiler-based approach to full memory access tracing, described in chapter 2. We show the benefits of using compile-time instrumentation to reduce the overhead and increase the amount of available information in memory access tracing.
- Access graphs – a model for capturing spatial locality properties of programs based on memory access traces, described in 3.2.
- A method for evaluating the potential for memory performance improvement from applying better data layout strategies. Our approach uses access graphs to derive better data layouts, which can be used either to estimate performance gains, or, more directly, to derive better layouts of data in storage.
- The eponymous Data-driven Spatial Locality technique, which uses random forest classifiers and data available from memory access traces to find correlations between data values and access patterns of a program. *This validates hypothesis 1.1.*
- An evaluation of the applicability of hint-based allocators to improve memory performance in programs, based on the knowledge extracted by Data-driven Spatial Locality technique. *Our evaluation confirms hypothesis 1.2 for one class of programs and reveals the obstacles to applying hint-based allocators more widely.*

Chapter 2

DINAMITE

2.1 Introduction

Memory performance is a limiting factor in many important programs. Traditional database systems, web servers, scientific algorithms and modern data analytics programs alike were observed to spend 50-80% of CPU cycles stalled on memory [7] [30]. That is, 50-80% of the time these programs are unable to commit any instructions due to outstanding long-latency memory accesses. Understanding and addressing the causes of these bottlenecks is of paramount importance. Performance improvements from a more efficient memory layout or improved locality of access are usually significant and in some cases reach an order of magnitude [45] [43] [61] [14] [32] [75].

At the same time, optimizing memory performance is notoriously difficult. Compiler optimizations can be effective when static analysis is sufficient to infer improvement opportunities. However, the scope of static optimizations is limited [18], partly because insight into a bottleneck can often be gained only during execution and partly because the compiler is limited in how it can change data structure layout, particularly with dynamically allocated data structures and in unmanaged languages. As a result, developers often resort to manually optimizing their data structures and algorithms, relying on tools for dynamic program analysis and memory profiling.

Unfortunately, most existing tools suffer from either lack of generality, portability, or flexibility. Conventional CPU profilers, such as perf [23], aim to identify source locations that generate the majority of cache misses, but because of skid effects in hardware counters [12], [38] or compiler optimizations, such as function inlining, this information is often imprecise or plain wrong. Cachegrind [56] accurately identifies source lines generating cache misses, but does not provide actionable insight that might lead the programmer to reduce them. Dprof [61] identifies data structures and fields that are responsible for cache misses due to sharing among threads, but it is not flexible enough to address other causes of poor memory performance and was designed specifically for the Linux kernel. Similarly, Memprof [43] focuses on identifying objects that cause remote accesses on NUMA systems, but

the implementation is Linux-specific, tied to AMD hardware and does not lend itself to other types of analyses.

To address this gap, we built DINAMITE – a toolkit for **D**ynamic **IN**strumentation and **A**nalysis for **M**ass**I**ve **T**race **E**xploration. DINAMITE uses compile-time instrumentation to inject tracing code into the program. At runtime, the program generates precise traces containing every memory access, its source location and the corresponding variable name, type and value. These traces are then used to perform various memory-related analyses, for example, identifying highest cache-miss offenders, locating hot and cold fields of a data structure, correlating locality of accesses with values of variables, detecting true and false sharing, building arbitrary models of memory access patterns, and many others.

The approach of using instrumentation and tracing is by itself not new; it is used in Pin [51], Valgrind/Cachegrind [56] and other similar tools. Its main downside is high runtime overhead and very large execution traces, which can reach hundreds of gigabytes even for small programs. However, for the very challenging task of memory performance debugging this approach is often the only practical option, because certain analyses, e.g., those relying on cache simulation, can be performed only with a precise execution trace.

Key contributions of DINAMITE are as follows:

- The instrumentation is implemented as a pass in LLVM [44], so it is applicable to any language with an LLVM front-end.
- Since the instrumentation is done at compile-time, the source-level debug information assigned to trace entries is precise and easy to extract.
- DINAMITE is specifically designed for memory access instrumentation. By instrumenting at compile time, we can embed all of the available debug information at no runtime cost. We show that shifting the overhead to compilation and using custom binary format and buffering in trace generation allows the runtime overhead to remain similar or smaller than state-of-the-art instrumentation tools such as Pin and Valgrind, while providing much more information about memory accesses.
- DINAMITE gives the user flexibility in how to handle execution traces. The traces can be stored in the file system, but if the user does not wish or cannot store these typically large traces, they can be analyzed on-the-fly using a streaming analytics engine like Spark Streaming [77] (or any other similar engine).

- DINAMITE is easy to extend with additional analysis tools. A developer can write a new tool with a few lines of Scala (if using DINAMITE with Spark) or any other language of choice. We target advanced developers who understand how software interacts with memory hierarchies of modern processors, so we wanted to give them ultimate flexibility in analysing memory traces.

We built three tools on top of DINAMITE. The first one produces variable names and source lines responsible for the highest number of cache accesses and misses. Using it, we reduced the last-level cache (LLC) miss rate of *429.mcf* from SPEC2006 by 55% and improved its performance by 12%. We also reduced the LLC missrate and improved performance of PARSEC's *fluidanimate* by 50% and 15% respectively.

The second tool implements Chilimbi's structure splitting algorithm [19]. Thanks to it, we reduced the LLC miss rate of SPEC2006's *429.mcf* by 60%, with the corresponding 20% reduction in runtime.

The third tool detects program variables that are heavily shared by many threads. This tool enabled us to detect a previously known performance bottleneck in WiredTiger, MongoDB's back-end key/value store [3] [4] [5]. Even though the bottleneck was already known and fixed before we created DINAMITE (in fact, this was one of the motivating reasons for DINAMITE), the original discovery took several weeks, while DINAMITE pin-pointed it in a few hours. Performance improvement of the read-only sequential LevelDB benchmark implemented over WiredTiger reached a factor of 20 for 32 threads.

The rest of the chapter is organized as follows. Section 2.2 provides an overview of DINAMITE design. Sections 2.2.2, 2.2.1 and 2.2.3 contain a detailed discussion of the log format, LLVM instrumentation pass and logging library. Section 2.2.4 discusses two implementations of analysis frameworks – one in native C++ and another one using Spark Streaming. Section 2.3 describes the tools we created with DINAMITE and evaluates them on three applications. Section 2.4 elaborates on possible avenues for future work.

2.2 System design

Our system is built from three components: an LLVM instrumentation pass, a collection of output logging libraries and an analysis toolkit. A system overview is shown in Figure 2.1. Different line types leaving the logging library show possible data paths within the system. A path taken by data depends on the type of analysis desired.

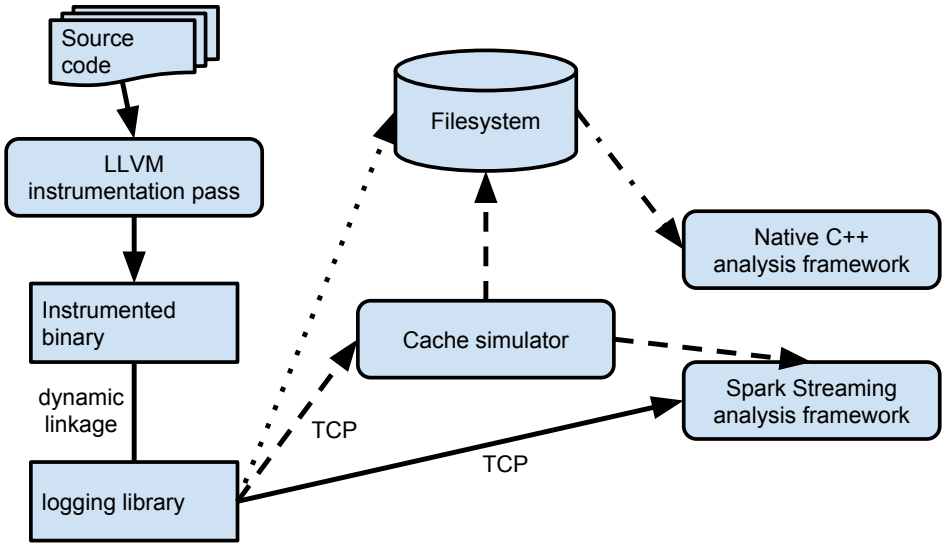


Figure 2.1: DINAMITE system diagram

Target applications are compiled with an LLVM [44] compiler configured to include our instrumentation pass. Configuration is trivial: it only requires changing the compiler invocation command. Most large software projects allow specifying the compiler command via an environment variable.

Our instrumentation pass instruments three types of events: function entry or exit, memory allocation, and memory access. For each event the instrumentation pass injects a function call to an externally linked logging library. The logging library, linked dynamically at runtime, will produce a log record of an event in binary or text format (described below). Log records are either stored in the file system or streamed over a socket. Stored traces can be analyzed using pre-packaged DINAMITE scripts written in Python or C++ (see Section 2.3), or the user can write her own tools using their language of choice. Log records streamed over the socket are processed by the Spark Streaming engine. DINAMITE includes several analysis kernels for Spark written in Scala; users can also write their own.

DINAMITE allows chaining analysis passes, similar to the Unix pipe command. For many of our analyses we stream the log data to a cache simulator tool (written in C++) that annotates each memory access log entry to indicate whether the access was a cache hit or a miss and forwards the annotated log entry to the Spark Streaming engine.

The rest of this section describes the system in more detail.

2.2.1 LLVM instrumentation pass

We chose to use the LLVM infrastructure for the implementation for the following reasons:

1. It can be used on programs written in any language supported by an LLVM front-end compiler. To date, those include, but are not limited to, C, C++, D, Haskell, Objective-C, Swift, Ruby, etc. There is even a compiler that translates Java bytecode into the LLVM intermediate representation. Given the popularity of LLVM, we can expect this list to grow in the future.

2. It lets us add instrumentation at the level of the intermediate representation (IR), which is more convenient than instrumenting a binary. LLVM IR is an assembly-like language that is more abstract than machine code (e.g., it assumes an unlimited set of registers). When IR is translated into binary, a single memory access can be expanded into multiple machine instructions, which can introduce noise into traces and make it difficult to attribute accesses to source code level constructs.

3. Full debug information is available at IR level. The front end we used, *clang*, embeds debug information into the IR as an abstraction of the DWARF format that is easy to parse with the tools provided in the LLVM framework.

Our instrumentation pass begins by crawling the IR debug metadata to extract information about complex data types (structs, classes, unions) and categorizes them by connecting their corresponding internal LLVM references with type and field names. This is necessary because of type aliasing. A single type in C or C++ can have multiple names because of `typedef`s. Without this metadata extraction, LLVM only knows about the original type definitions, but IR instructions may contain references to different names for the same type. We store all the type alias information in map-like data structures within the pass.

The core of the instrumentation pass iterates over the current module and visits each function, each basic block and each instruction within it.

For each encountered function, it places a *function begin* log call at the beginning of its first basic block, and a *function end* log call at the end of each basic block that ends the function.

Memory allocation functions are treated separately. Our pass must first recognize whether a function is a memory allocator and then gather the

information about the allocated type, size and address. For each called function, our instrumentation checks the function's name against the list of known allocator functions. We generalize allocator functions as functions that take two arguments: *number of elements* and *size of a single element*, and output the address that points to the start of the allocated region. This model encompasses all allocation library APIs that we have encountered, and, combined with type information available through LLVM, contains all the relevant information that describes an allocation.

The list of allocators is provided in a separate file, where each entry is described with a function name, and three indices indicating the position of all the relevant fields (the number of elements, the size of the element and the allocation's base pointer) in the argument list. If the allocation address is the function's return value, its index will be set to -1. Standard allocation functions (such as `malloc` and `calloc`) are included in the configuration file that comes with our pass. If the program uses any non-standard allocator functions, the user must add them to that file. The pass places an allocation event log call after each call to an allocator function.

Strings written to the log are encoded with a unique integer identifier to preserve space. The integer-to-string mappings are placed into JSON documents created by the instrumentation pass. The mappings must be consistent across modules, but LLVM compiler passes do not preserve any state across modules. Therefore, we load the JSON mappings before compiling each module compilation and write back any updates at the end.

For ease of use with C and C++ projects, our instrumentation pass gets registered with LLVM's pass manager for standalone clang invocations. As clang supports most of GCC's compilation flags, this makes integration of our instrumentation into existing projects in most cases as easy as changing the compiler invocation variable.

2.2.2 Log format

All log events contain a field for a thread identifier. We limit this field to 8 bits to conserve space, but it can be easily expanded if needed. Distinguishing between 256 unique threads was sufficient for our case studies.

Allocation and access events share fields that contain the file name, the line number and the column number that correspond to the event's source code location. Allocation events additionally contain the base address of the allocated memory region, the size of a single allocated element, the number of allocated elements and their type.

Access events contain the accessed address, the type of the access (read

or write), the name and type of the variable corresponding to the access, and the value at the accessed address.

Function events contain the event type (entry or exit into the function) and the function name.

Depending on the configuration, log records can be produced in text or binary format. In text format, each field of the entry is printed to a file, separated by a delimiter character. In binary format, different types of events are contained in a parent `logentry` structure. The `logentry` structure contains a type field that differentiates the payload as either a function, access or allocation event. The payload is a union between the corresponding three log entry types. In the current version of DINAMITE, each log entry takes 48 bytes total.

Log format involves a surprising trade-off between performance and log size, which we evaluate in the next section.

2.2.3 Logging libraries

Instrumented programs do not contain any logic for producing log records. Instead, they contain calls to the externally linked logging library. Our implementation contains three different library versions based on log format and output destination: text-to-file, binary-to-file and binary-to-socket.

The effect of log format on log size

In our implementation, each binary log record takes up 48 bytes of storage. Text entries are variable in size and depend on the number of characters needed to encode all the values. The two extremes of an entry size in text format are:

- Minimum: 18 bytes. Each field can be encoded with a single digit, with added single character delimiters.
- Maximum: 77 bytes. Each field has the maximum value for its storage type in binary format.

The reality is somewhere in-between, as shown in Table 2.1, which compares log sizes in text and binary format for SPEC2006 *429.mcf*, with 4.5 billion memory accesses. Text format generates smaller logs; log size is important, because real workloads generate hundreds of gigabytes of logs. At the same time, using text format results in a much higher performance overhead (evaluated in the next section). Since we can forego storing large logs by relying on DINAMITE's streaming model we always use DINAMITE with the binary log format to avoid the overhead.

Table 2.1: Log size comparison in 429.mcf

Number of accesses:	∼4.5 billion
Binary log size:	205GB
Text log size:	172GB

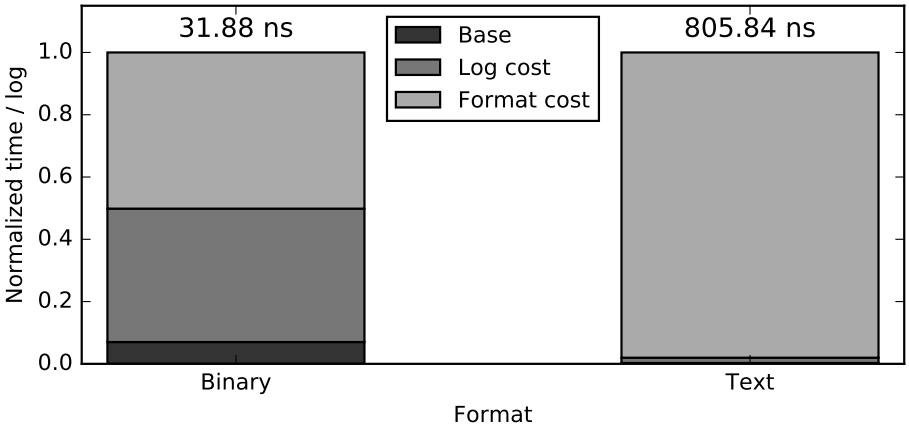


Figure 2.2: Cost breakdown of text and binary formats for 429.mcf, per single log entry

The effect of log format on performance

For a detailed insight into the overhead of running the instrumented binary, we break it down into the following components:

- *base cost*: the cost of executing the uninstrumented code
- *log cost*: the cost of invoking the logging library function
- *format cost*: the cost of preparing the log entry for writing
- *output cost*: the cost of writing the log entry

Table 2.2 and figure 2.2 show the broken-down cost of the instrumentation for *429.mcf*. We report all costs except the output cost, because it depends on where we write the log data; the output costs for different output destinations are reported later in this section.

Log cost is fixed and must be invoked for every instrumented event. The only way to reduce this cost is to avoid instrumenting certain events altogether, according to a user-defined criterion at compile time. For example,

Table 2.2: Cost breakdown of text and binary formats for 429.mcf, per single log entry

Format	Binary	Text
Base	2.33 ns	
Log cost	13.66 ns	
Format cost	15.99 ns	789.95 ns
Total time (no output)	31.98 ns	805.94 ns

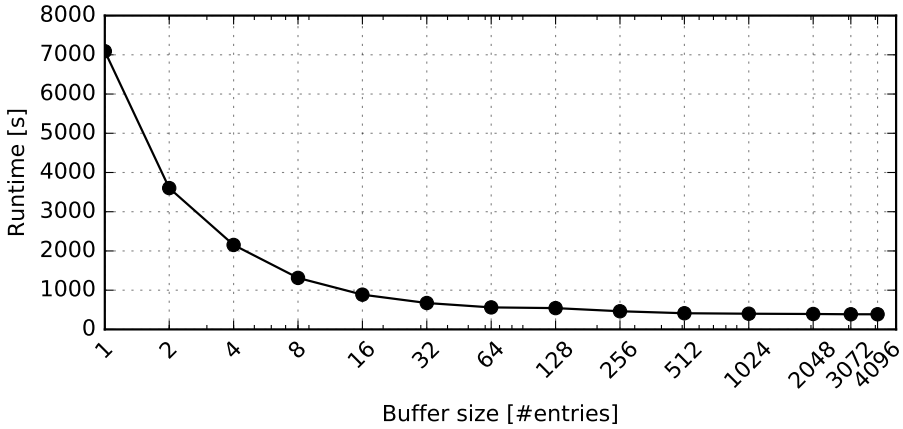


Figure 2.3: Impact of buffering on performance of 429.mcf

if we are not interested in exploring the entire memory access trace of a program, but only accesses to a single type, we can tell the compiler to instrument only those. Similarly, we can limit instrumentation to certain functions. Instrumenting isolated data structures or types can discover some memory access patterns, but this kind of filtering is not appropriate for understanding whole program memory behavior or for any analysis involving cache simulation.

Format cost is the cost of packing the log data according to the specified format. It is dominant for the text format, because formatting strings is a very expensive operation, relative to producing a binary record. Text format suffers a 25× higher run time relative to the binary format. That is why we always resort to using the binary format in our experiments, despite its higher storage overhead.

Output cost is the cost of writing the log records into a file or sending them over a socket. Besides the cost of accessing the storage medium, it

requires a system call. We mitigate this overhead via buffering. Figure 2.3 shows the effects of different buffer sizes on the runtime of 429.mcf in the binary format that writes to a RAM disk. By increasing the output buffer size, performance improvement reaches its maximum at around 20× over the unbuffered version. In the rest of our measurements, we used the output buffer of 4096 entries.

Table 2.3 compares the performance of *429.mcf* instrumented with DINAMITE against slowdowns of two major instrumentation frameworks: Intel Pin and Valgrind. Valgrind performance degradation reported here is obtained from Nethercote et al. [56], and refers to Valgrind's MemCheck tool which performs memory error checking with a summary output at the end of execution. This is not a fair comparison to access instrumentation, but to the best of our knowledge, a Valgrind tool comparable in functionality with DINAMITE is not available. Numbers for Pin were obtained from the supplied `pinatrace` tool, output to RAM disk. In Table 2.3, the slowest version of DINAMITE without analysis instruments each access with full debug information (as described in section 2.2.2) and outputs it to a RAM disk filesystem in binary format. Even at this level of detail and with full output enabled, DINAMITE is only 60% slower than Valgrind's MemCheck and almost 10x faster than the comparable access instrumentaion in Pin. Even when using the Spark analysis pipeline, DINAMITE is only 35% slower than `pinatrace`.

Table 2.4 compares the running times for executing *429.mcf* with different variants of log formats and outputs. Note that the text-formatted output makes the instrumentation run very slow: 33× slower than using the binary format. Sending the trace over a TCP socket to `netcat` is faster than writing it to a RAM disk. However, introducing Spark Streaming into the pipeline makes the TCP streaming execution 15x slower. Optimizing this would require a detailed analysis of Spark's data receiving system and is left for future work.

Our design decouples the generation of log records from their processing. Alternatively, embedding analysis logic into the logging library is also possible. We opted against it for the following reasons:

- Instrumented programs share heap with the logging library. Adding significant bookkeeping data structures to the heap could affect the placement of the program's data and diminish the accuracy of traces.
- Decoupling analysis from logging allows for flexibility in the languages and frameworks used for analyzing memory traces

Table 2.3: Instrumentation overhead comparison - 429.mcf

Framework	Slowdown
Pin (pinatrace output to RAM disk)	354x
Valgrind (MemCheck)	22x
DINAMITE (empty instrumentation)	7x
DINAMITE (binary format, no output)	14x
DINAMITE (binary format, output to RAM disk)	36x
DINAMITE (Spark analysis)	537x

Table 2.4: Logging library performance - 429.mcf

Version	Destination	Time [s]	Slowdown
Uninstrumented	nil	10.05	1x
Text (unbuffered)	RAM disk	11820	1176x
Binary (file) (buff.)	RAM disk	360.09	36x
Binary (file) (buff.)	Hard disk	1426	142x
TCP (buffered)	netcat >/dev/null	339.12	34x
TCP (buffered)	Spark (access count)	5400	537x

2.2.4 Analysis toolkit

The analysis toolkit consists of two different frameworks for writing log processing applications. Logs recorded to a filesystem are processed with the native analysis framework written in C++. The framework includes support for parsing logs and allows the user to easily extend the analysis by writing a new C++ class. Alternatively, the users could write their own parsing and analysis tools in any language of choice. Logs streamed over a TCP socket are processed live with Spark Streaming drivers. Similarly, the user could configure DINAMITE to use any another system to ingest or analyze streaming logs (e.g., Kafka, Google Dataflow). Our toolkit includes a simple cache simulator program, which processes and annotates streamed log records with cache hit/miss indicators. Next, we describe these components in more detail.

Native analysis framework

The C++ framework provides support for writing arbitrary analysis kernels. To write a new kernel, the programmer must extend the `TracePlugin` class (shown in listing 2.1).

Listing 2.1: Trace plugin base class

```
1  class TracePlugin {
2  ...
3   protected:
4   NameMaps *nmaps;
5   TracePlugin(const char *name);
6   public:
7   virtual void processLog(logentry *log) =0;
8   virtual void finalize() =0;
9   virtual void passArgs(char *args) =0;
10 };
```

The framework reads log records into a buffer and passes each log entry to the chosen plugin by invoking its `processLog(logentry*)` method. At the end of the log file, the framework calls the plugin's `finalize()` method, which is used for writing the output of the analysis.

Spark Streaming analysis framework

To analyze streamed log records with Spark Streaming, the programmer must write an analysis kernel in Scala. To this end, our framework provides a custom Spark Streaming `Receiver` class and a log converter. A receiver accepts batches of log events in binary format over a TCP socket and stores each separate log entry in its associated `StreamingContext`.

Listing 2.2 shows a Spark Streaming kernel for counting the number of memory accesses per variable. To get useful information out of the entries, the incoming `DStream` is routed through a `map` operation which invokes our `LogConverter` class on each separate entry. `LogConverter` unpacks and outputs log data as Scala classes, with the distinction between function events, allocation events and access events. To get a `DStream` of instances of a certain event type, logs are filtered with a class matching operation. These events are then mapped to `(varId, 1)` pairs, and reduced by summing over variable IDs. Persistent state is updated by invoking Spark Streaming's `updateStateByKey()` operation. A custom update function, omitted in our listing for brevity, updates the counts by summing new results with the previous state. Results are then output to the console or the filesystem.

Listing 2.2: Example Spark Streaming kernel

```
1  def main(args: Array[String]) {
```

```
2     val sparkConf = new SparkConf()
3       .setAppName("AccessCounter");
4     val ssc = new StreamingContext(sparkConf,
5       new Duration(1000));
6     ssc.checkpoint("/checkpoints/");
7
8     val logs = ssc
9       .receiverStream(new LogReceiver(9999))
10      .map(rawlog =>
11      LogEntryReader.extractEntry(rawlog));
12
13    val counts = logs
14      .filter(log =>
15        log.isInstanceOf[AccessLog])
16      .map(access =>
17      (access.as(...)[AccessLog].varId, 1L))
18      .reduceByKey(_+_)
19      .updateStateByKey(sumUpdater);
20
21    counts.print();
22
23    ssc.start();
24    ssc.awaitTermination();
25  }
```

Integration with Spark Streaming gives the programmer access to the full set of Spark Streaming operations and can process logs as they are output from a live running program.

Cache simulator

For detailed analysis of program cache behaviour, we wrote a simple cache simulator, which is placed as an intermediate step between the generation of the log output and the analysis framework (or the filesystem, if we are saving the logs for offline processing).

We simulate a single-level cache, typically configured with parameters reflecting a last-level cache on our target system. The cache simulator accepts log entries over a socket, much like the Spark Streaming analysis framework. It annotates each memory access with an indicator whether this was a cache hit or a miss. The annotated logs are passed on to either the analysis framework, or stored in a filesystem for offline processing.

In our evaluation of the system, we found that having cache behaviour information was essential for identifying certain optimization opportunities.

Table 2.5: 429.mcf top miss offenders

Variable Name	File	Line	Miss count
arc.ident	pbeampp.c	167	45247271
node.orientation	mcfutil.c	85	1104784
node.basic_arc	mcfutil.c	86	988543
arc.cost	mcfutil.c	86	767273
node.potential	pbeampp.c	170	235696

2.3 Evaluation

In this section we describe three tools that we built using DINAMITE and demonstrate how they guided our optimization of three applications: SPEC's *429.mcf*, PARSEC's *fluidanimate* and *WiredTiger*, a MongoDB's key-value store.

All experiments described in this section were performed on machines listed in the Appendix.

2.3.1 Identifying cache offenders

> "For large working sets it is important to use the available cache as well as possible. To achieve this, it might be necessary to rearrange data structures. While it is easier for the programmer to put all the data which conceptually belongs together in the same data structure, this might not be the best approach for maximum performance."
>
> Ulrich Drepper, 2007[27]

This tool identifies program variables and source lines that generated the most last-level cache accesses and misses. It works as follows:

429.mcf

429.mcf performs single-depot vehicle scheduling using the network simplex method. The implementation represents nodes and arcs in the network as C `structs`. In the benchmark description the author mentions reordering fields of both node and arc structs in an attempt to reduce cache misses and improve performance [34]. Nevertheless, DINAMITE enabled additional optimizations.

Table 2.5 shows the output of the cache-offender tool. We notice that a disproportionate number of misses are being caused by the `ident` field of

the `arc` struct, more than four times as many as the second most accessed field, `node.orientation`.

Upon closer inspection, we noticed that all the `arc` structures are allocated as a single large array, even though they represent nodes in a linked data structure. The majority of accesses to `arc.ident` were made within a single loop (shown in Listing 2.3). The loop iterates over the `arc` array until it finds a match, and only then accesses its other fields.

Every time `arc.ident` was accessed, the corresponding cache line was filled with other fields, most of which were not used before the cache line was evicted. These data layout and access pattern waste cache space and memory bandwidth. We addressed the problem by restructuring the array of `arc`s from the *array of structures* layout into the *structure of arrays*.

Listing 2.3: 429.mcf pbeampp.c excerpt

```
165 for( ; arc < stop_arcs; arc += nr_group )
166   {
167     if( arc->ident > BASIC )
168       {
169         /*red_cost = bea_compute_red_cost(arc);*/
170         red_cost = arc->cost - arc->tail->potential
                + arc->head->potential;
171         if( bea_is_dual_infeasible( arc, red_cost )
                )
172           {
173             basket_size++;
174             perm[basket_size]->a = arc;
175             perm[basket_size]->cost = red_cost;
176             perm[basket_size]->abs_cost = ABS(
                    red_cost);
177           }
178       }
179   }
```

Our modifications brought a 55% reduction in LLC misses, and a 12% improvement in the overall runtime.

fluidanimate

Fluidanimate is an Intel *Recognition, Mining and Synthesis* application that uses the Smoothed Particle Hydrodynamics method to simulate an incompressible fluid. It uses the Navier-Stokes equation to derive fluid density fields. It is included in the PARSEC 3.0 benchmark suite because of the increasing significance of physics simulation in video-game programming and

Table 2.6: CSV output of the miss summary tool for fluidanimate

Variable name	File	Line	Miss count
Cell.next	pthreads.cpp	530	184496
Vec3.x	./fluid.hpp	354	95682
Cell.next	./fluid.hpp	404	73800
Vec3.x	./fluid.hpp	346	67327
Vec3.x	./fluid.hpp	355	66657

real-time animation domains [11].

Profiling *fluidanimate* with `perf` [23] showed that it has a high LLC miss rate of 30% on our system. We instrumented the program using DINAMITE and ran it through the cache offender tool. Table 2.6 shows output of the tool: variable names and source lines responsible for the most cache misses. The top cache offenders are `Cell.next` and `Vec3.x`. The names of the structs and fields suggest that a `Cell` is an element of a linked collection. Listing 2.4 shows the code excerpt pointed to by the output of our tool. We can immediately see that the code generating misses is a traversal of grid of `Cell` structures in which only the `next` field is touched.

Looking at the definitions for `Cell` and `Vec3` types we can see that `Cell` represents a linked list of containers for arrays of `Vec3` structures that contain three-dimensional vectors. The arrays themselves are contained within the `Cell` struct in their entirety. The total size of a `Cell` struct with the payload was 896 bytes, making a single instance span 14 cache lines.

This data layout is poorly optimized for traversing lists of `Cell`s, because each new `Cell` access generates a cache miss. Our idea, therefore, was to allocate the `Cell`'s payload, which is rarely touched, separately from the rest of the structure. The structure would then include a pointer to its payload; since each `Cell`'s payload consists of multiple arrays, adding a layer of indirection to access the payload would not be much of a penalty. Allocating the `Cell` payload separately brings the size of the structure down to 16 bytes . Since consecutive calls to a memory allocator function for a variable of the same size will return near consecutive addresses in most standard libraries, consecutive `Cell`s will be allocated close together, and several of them will fit into a single cache line.

Listing 2.4: fluidanimate pthreads.cpp code excerpt

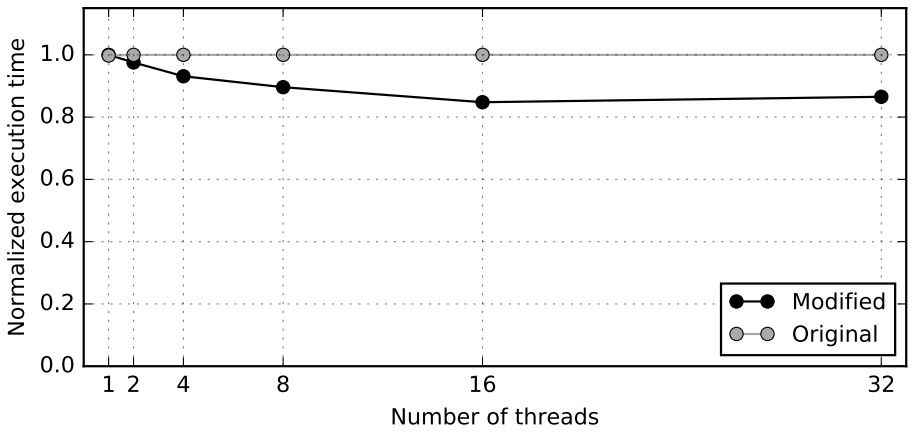


Figure 2.4: Scaling improvements in PARSEC3.0 fluidanimate application

```
522 void ClearParticlesMT(int tid)
523 {
524   for(int iz = grids[tid].sz; iz < grids[tid].ez;
          ++iz)
525     for(int iy = grids[tid].sy; iy < grids[tid].ey;
            ++iy)
526       for(int ix = grids[tid].sx; ix < grids[tid].
              ex; ++ix)
527         {
528           int index = (iz*ny + iy)*nx + ix;
529           cnumPars[index] = 0;
530           cells[index].next = NULL;
531           last_cells[index] = &cells[index];
532         }
533 }
```

This change brought a 50% reduction in the LLC cache miss rate and a 15% reduction in runtime with 16 threads (see Figure 2.4).

An interesting observation is that `Cell`s were allocated in the original implementation to be cache-aligned and padded to a fill the entire cache line, indicating a prior effort to make better use of the cache hierarchy. However, with DINAMITE we discovered that making the `Cell` struct larger by padding actually hurt performance on our system. Intricacies of modern multi-core memory hierarchies can mislead even very experienced programmers. Powerful performance analysis tools are thus crucially important.

2.3.2 Structure splitting

Our structure splitting tool is based on the class splitting algorithm proposed by Chilimbi et al. for Java programs [19]. The algorithm analyses how the program accesses the members of a class to determine if a class is fit to split into two separate classes. Splitting classes is motivated by the idea that *hot* fields, or fields that are accessed significantly more than *cold* fields, should be placed in a separate class so that more hot data can be packed into a single cache block. To access fields that are considered cold, the hot class includes a pointer to the cold class. (This is similar to the optimization that we applied to *fluidanimate*).

The algorithm begins by identifying *live* classes. A class is considered *live* if it is accessed more than a certain threshold, and only live classes are considered for splitting. Next, fields in live classes are marked as hot or cold depending on how many times their respective class is accessed. If a field is accessed significantly more than other fields, it is considered hot. Full details of the algorithm are described elsewhere [19].

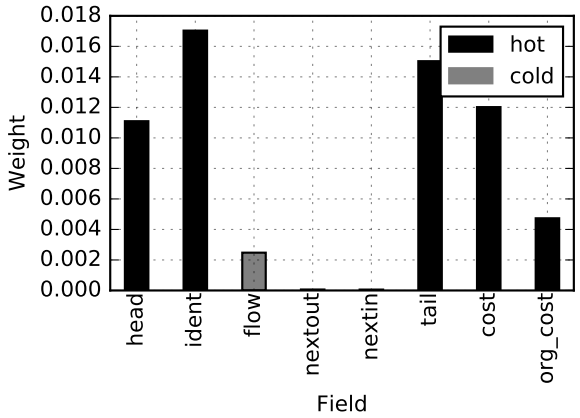
Chilimbi et al. implemented the splitting algorithm for Java classes, using the JVM for access statistics and a Java byte-code instrumentation tool BIT [47]. We implemented the splitting algorithm in DINAMITE, making it accessible to a wider range of programs, including those written in unmanaged languages. The tool works as follows:

The Spark Streaming driver receives access logs from the instrumented binary and produces the list of variables and corresponding access counts. Then a Python script generates a chart for each live structure showing weights assigned by the algorithm for individual fields; black bars for hot fields and gray bars for cold fields. Programmers then split their structures according to the hot and cold fields in the chart.

Figure 2.5 shows the chart produced by the structure splitting tool for the `arc` struct in *429.mcf* as well as the modified `arc` struct code. Similarly struct `node` (not shown) was another live struct with both hot and cold fields. Splitting hot and cold fields in these structs delivered 20% speedup and reduced the LLC miss rate by 60%, as measured with `perf`.

2.3.3 Shared variable detection

On machines with even a handful of cores, variables updated by multiple threads can quickly become a scaling bottleneck [14], even if these variables are not protected by a lock or accessed via atomic instructions [26]. Repeatedly updating a shared variable from different cores stresses the coherency




```
struct cold_arc {
  flow_t flow;
  arc_p nextout;
  arc_p nextin;
};

struct hot_arc {
  cost_t cost;
  node_p tail, head;
  int ident;
  cost_t org_cost;
  struct cold_arc *ca;
};
```

Figure 2.5: Tool output and modified code for structure splitting of 429.mcf

protocol and can slow down the program by an order of magnitude relative to a sharing-free execution. Tools for detecting shared variable bottlenecks do exist, but they are hardware-specific (e.g., Intel's VTune [53] works only on Intel machines, DProf [61] and Memprof [43] work only on AMD hardware) and can be non-trivial to set up (DProf and Memprof require changing the kernel). DINAMITE is easily extended to detect shared variable bottlenecks on any binary that can be compiled with LLVM.

To demonstrate, we created a simple tool that we then used to quickly find a known scalability bottleneck in WiredTiger [3] [5], a MongoDB storage engine [4]. To truly test the experience of creating new tools for DINAMITE, the student who created the shared variable detection tool was *not* informed what variables and source locations triggered the bottleneck; he was only advised that the bottleneck exists and provided the instructions for running the problematic workload.

The engineer who originally diagnosed the scalability issue took about week to do so after observing poor performance; she used Memprof, which required communication with its authors and changes to the kernel. Even though the changes were simple, they would likely be considered "beyond the call of duty" by many developers. The DINAMITE tool took several hours to create by a person familiar with the overall framework and consisted of two simple Spark Streaming kernels and a Python script.

The first and second kernel identify top shared variables. The second kernel processes the execution traces again, looking only for frequently shared variables and collects the source locations where the accesses are made. The

tool could be structured with only a single kernel that both identifies the top shared variables and records the source lines, but we found that having two kernels is simpler and results in better performance of Spark Streaming.

The first kernel translates memory access log entries into (*accessed address*, *variable identifier* and *thread identifier*) tuples. Each tuple acts as the key in *map-reduce* transformation that produces a list of variable-identifying tuples and the corresponding total memory accesses. The result is stored in a persistent table.

Next, a Python script reads that table and transforms the data into a dictionary, where each accessed address serves as a key and the corresponding value contains the variable identifier and access counts performed by each thread. The script filters the results according to the following criteria:

- It removes all entries accessed by only a single thread
- It removes all entries that are not uniformly shared by threads. We define uniform sharing as follows:

  Let A_{sorted} be a list of all the per-thread access counts for the address, sorted in descending order, zero indexed.

  if $A_{sorted}[0] < 2 * A_{sorted}[1]$ the sharing is uniform.

The output is then sorted in descending order by the total number of accesses. Table 2.7 shows the first five entries of the output generated for the LevelDB sequential read benchmark over WiredTiger (release 2.6.0) executed with 32 threads[3], which triggered the bottleneck. The top offender is the field `v` in `__wt_stats` struct.

These results only point to the variable responsible for shared accesses. To find the root cause, we use the second Spark kernel to find the source location where the accesses are performed. The second kernel is very similar to the first, except it discards all the log entries that do not correspond to the top shared variable, and keeps the source lines where the accesses occurred. We sort the results by the number of accesses in descending order.

Listing 2.5: WiredTiger shared variable analysis result (JSON)

```
1 {
2   "threadcount": 18,
3   "totalcount": 311881,
```

[3]The connection structure is shown as accessed by more than 70 threads because the benchmark creates and tears down additional threads before the measured run.

Table 2.7: Most accessed shared variables

Address	# Accesses	# Threads	Variable
0x64D900	*42495568*	*32*	*__wt_stats.v*
0x64D1A4	26183326	74	__wt_connection_impl
0x64E0EC	7233836	72	__wt_connection_impl.N/A
0x64D100	4786616	36	__wt_txn_global.states
0x64D540	4786370	34	__wt_stats.v

```
4    "threads": [
5     [
6      156,
7      19492
8     ],
9     [
10     163,
11     19520
12    ],
13    ...
14    ],
15    "file": "wiredtiger/build_posix/../src/btree/
          bt_curnext.c",
16    "line": 446,
17    "variable": "__wt_stats.v"
18   }
```

Listing 2.5 shows the first entry of the output (redacted for brevity), which correctly identifies the source location responsible for the bottleneck. The fields in the output JSON document are self-explanatory apart from the `threads` list, which contains `(thread id, thread access count)` pairs. It turns out that this sequential read-only workload suffered from scalability problems, because threads were incrementing a shared statistics counter after each read operation. This problem was later fixed by implementing per-thread statistic buffers.

Figure 2.6 shows the performance impact of the bug on Machine A (described in the appendix). A seemingly benign counter increment, which takes negligible time in a single-threaded execution, quickly escalates into a huge scaling bottleneck with as few as four threads and slows down the workload by an impressive factor of 20 with 32 threads. Previous work reported similar performance impact of shared variables on multicore machines [26]. With the increasing core counts on new hardware the importance of tools that enable productive memory performance analysis will continue to grow.

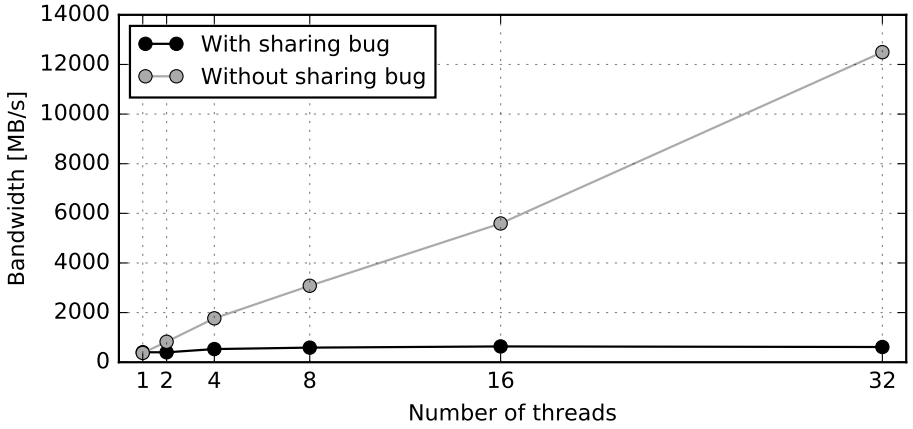


Figure 2.6: Scaling improvements WiredTiger after removing the shared variable bug

2.4 Future work and conclusions

We described the implementation of DINAMITE and discussed the performance implications of our design choices: a fine-grained compiler-based instrumentation and a flexible analysis framework. Our detailed breakdown of the costs involved in doing instrumentation of memory accesses leads us to conclude that this kind of design is not only feasible, but allows for a great level of detail in the generated traces, while keeping the slowdown comparable to the state-of-the-art instrumentation frameworks. We introduced a novel fusion of instrumentation and stream processing that eliminates the need for storing traces and provides an easy to use Spark Streaming API for analysis purposes. Finally we demonstrated the utility of DINAMITE by performing three different types of analysis that were difficult, impossible, or constrained to a certain OS/hardware platform with the previously available tools.

In future work, we plan to expand on the kinds of analysis that can be done on access traces using a streaming framework. Spark Streaming currently buffers incoming log records and processes them at the expiration of a configurable timeslice. In our experience, the timeslice must be rather large (e.g., one second) to avoid performance problems with Spark, but with such a large timeslice the batches contain hundreds of thousands of accesses. Adding support for a framework that is able to process a batch of records after it accumulates a specified *number* of records would let us have

finer granularity in our analysis. This kind of setup would allow discovering access patterns within small windows of time, which is important for certain optimizations.

Further, we plan to explore and optimize the slowdown of DINAMITE when using the full analysis pipeline with Spark Streaming. Finding the best way to integrate the log generation and analysis is an important factor in improving the overall productivity of engineers using our system.

Finally, our implementation of the cache simulator is rather simplistic. Adding a multi-level cache simulator such as Dinero IV [28] and adding more cache information to the logs would help improve understanding how different data organization and access patterns affect program efficiency.

Chapter 3

Data-driven spatial locality

3.1 Introduction

Memory wall is the phenomenon where the cost of memory accesses exceeds the cost of non-memory instructions to the point that the program spends most of its CPU time waiting on memory. Wulf and McKee proposed that modern software would hit the memory wall in the beginning of this millennium [73]. Ailamaki, DeWitt, Hill and Wood showed in 1999 that database systems spent 20-50% of CPU time waiting on memory [7]. For modern "cloud" workloads this figure is 50% on avearage, and reaches 90% for OLTP benchmarks [30]. Caches mitigate memory latency, but it is believed that they will never catch up with the voracious appetite of modern applications [30].

To navigate the limitations of hardware, programmers invest substantial effort into software optimizations aimed at reducing cache miss rates. One family of such optimizations is about rearranging the program data in memory in order to improve spatial locality. Spatial locality occurs when data items that are accessed close together in time also happen to reside close together in memory. Hardware caches fundamentally rely on spatial locality for efficient operation. Finding an optimal arrangement of objects in memory is NP-hard. A guiding principle used in prior work on memory layouts is ***to put objects that are frequently accessed together close to each other in the address space***. Literature review has revealed that these optimizations are largely manual and require deep understanding of the program's algorithms and data structures, making many of them the subjects of top-tier publications [10, 17, 25, 36, 37, 42, 50, 65, 67, 69, 75, 76]. These algorithms deliver significant performance improvements, but are very difficult to implement.

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

that operates on many objects in memory and automate the extraction of knowledge needed to derive new layout strategies.

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

The contributions of this work are as follows:

1. Access graphs: A novel way of representing memory access patterns within a program. Access graphs reveal which objects in memory are accessed contemporaneously. They are computed from allocation and access traces. By using our proposed analysis techniques, programmers can automatically extract expert knowledge of access patterns that is key in most prior work on data layouts.

2. Hierarchical Memory Layouts (HML): A new algorithm that uses access graphs to automatically derive improved data layouts for memory intensive programs. HML combines prior work in graph community detection and linear arrangement in a novel way in the context of spatial locality optimization. Using HML, programmers can get an estimate of how much room for improvement there is in a program's data layout. In certain workloads, HML can be used directly to recompute layouts of data in storage.

3. Data-Driven Locality: A novel application of Random Forest Classifiers to detect correlations between memory access pattern and data properties. We use random forests to learn which features of data itself can be used at runtime to group allocated objects, with the goal of improving spatial locality. We use these techniques to automatically infer expert knowledge from prior work on data structure layouts

(graphs and meshes), and expand the application to red-black trees. To evaluate the performance gains we built Tidy, a hint-based allocator wrapper that lets us use objects' data field values to guide allocation at runtime.

Figure 3.1 shows a detailed workflow diagram for the techniques presented in this paper. We will not go into much detail about each of the nodes in the diagram, but we encourage the reader to refer back to it as they read through sections 3.2-3.4. It is divided into three different stages, outlined with dotted lines. The first stage shows the process of *access graph creation*, which is described in detail in §3.2. The second stage is *layout performance evaluation*. In this stage, we evaluate the potential for performance improvement from changing the data layout of the program. Layout performance evaluation is covered in §3.3. Finally, if the programmer deems it worth to change the data layout based on cache simulation results, they proceed to the third stage – *Data-Driven Spatial Locality*, described in §3.4. In this stage, we use machine learning techniques to discover data features that can be used as hints for Tidy, our allocator wrapper.

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

The rest of this chapter contains evaluation and discussion of our results in §3.5.

3.2 Access Graphs

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

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

Besides accesses to dynamically allocated (heap) objects, memory access traces contain accesses to global variables and local (stack) variables. We focus on large data structures that generate many cache misses – they are

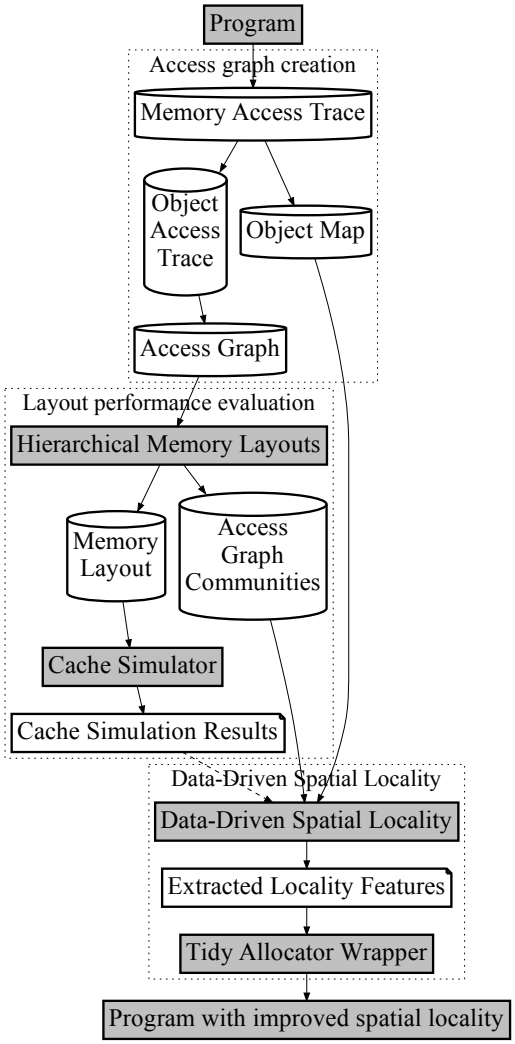


Figure 3.1: Workflow diagram

most likely to consist of dynamically allocated objects. Therefore, we filter data objects of other kinds from the trace.

Definition 3.2.1 (Object Access Trace) *Object Access Trace is a filtered form of a program's dynamic memory access trace. It is obtained by first removing all stack and global accesses from the trace. Next, all the accesses that target heap objects are replaced with accesses to the target objects' base addresses.*

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

Definition 3.2.2 (Object Map) *An Object Map is a hash map of all the allocated objects. It is generated by processing the original memory access trace, before converting it into an Object Access Trace. When an object is accessed, the offset at which the access was made is recorded in the map, along with the type of access (read/write) and the value read/written. Object Maps give information about the type of object and the contents of all of its data fields.*

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

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

Definition 3.2.3 (Access Graph) *An access graph is an undirected graph where there is a vertex for every dynamically allocated object in the Object Access Trace. Two vertices are connected by an edge if there are at least two*

contemporaneous memory accesses to the objects represented by these vertices. Two accesses are considered contemporaneous if they occur within C_L memory accesses of one another. C_L is called a locality constraint. Whenever we detect the first contemporaneous access we create an edge between the two vertices and assign it the weight of one. Whenever we detect another contemporaneous access to vertices already linked by an edge, we increment the edge's weight by one.

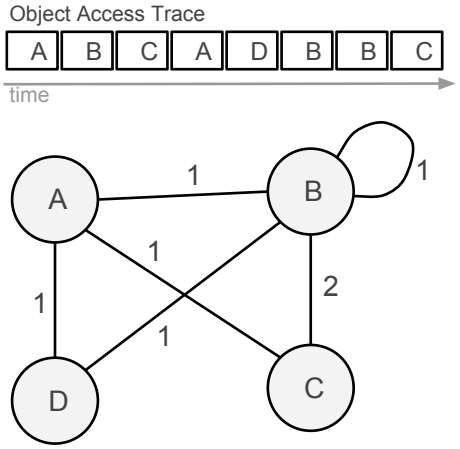


Figure 3.2: Access graph example

Figure 3.2 shows an example of an Object Access Trace with the corresponding access graph for $C_L = 2$. Note that D and C have no edge between them because there are no contemporaneous accesses to them in the trace. Objects B and C have an edge with the weight of two, because there are two contemporaneous accesses to them in the example trace.

Choice of C_L value for computing access graphs has two aspects that should be considered: *computational cost* and *captured information*. From the computational cost point of view, the C_L value tells us exactly how many edge additions/updates we have to perform for each access in the Object Access Trace. Because of this, C_L should be as low as possible, while retaining important information in the access graph. As for the second aspect, we empirically tested higher values for C_L for Hierarchical Memory Layout computation (described in §3.3), but did not observe a significant improvement in the quality of results. For our analysis techniques, we used $C_L = 2$, meaning we only consider two immediately adjacent accesses in the

Object Access Trace. Using access graphs for other purposes may require reconsidering this choice.

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

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

Given an access graph, how do we use it to improve spatial locality? The graph tells us which objects are accessed contemporaneously. How do we translate this information into a more efficent program? To answer this question, we will break it down into two parts. First, we will find out if there is a potential to reduce the cache miss rate by using a different data layout. Given an Object Access Trace and an Object Map, we will assume that we can rearrange the memory addresses of the objects in any way we like (without worrying how this could be achieved in practice), replay the access trace in a simulator and evaluate the cache miss rate resulting from the new layout. We compute the new layout using the new Hierarchical Layout Algorithm that we describe in §3.3. Although this is not a concrete solution that a programmer can use directly, evaluating layout changes in an abstract way will help us understand if there is a potential to improve performance by changing the layout.

The second part of the question asks how we can improve the data layout in a concrete program. Our work on Hierarchical Memory Layouts in §3.3 describes the process of obtaining good memory layouts that can be used directly to reorder data in storage. Section §3.4 presents a machine learning technique that trains on the Object Map and discovers the properties of the data objects available at object allocation time that can be used to guide object placement in memory at runtime.

3.3 Hierarchical Memory Layout

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

Community detection algorithms detect groups of nodes in graphs such

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

Our Hierarchical Memory Layout algorithm extends a multilevel community detection algorithm by Blondel et al [13]. Multilevel community detection starts with every graph node representing its own community. It expands communities by adding close neighbours into them, making sure that the change will result in higher inter-cluster separation, and intra-cluster proximity (together called *modularity*). When such a change is impossible, the algorithm stores the community assignment and transforms the graph by fusing all nodes within a community into a single node, and aggregating all edges to other such community nodes. This process is repeated on the transformed graph until there are no changes that can be done that improve modularity.

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

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

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

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

These rules are in line with the optimization objective of the Minimum Linear Arrangement problem (MinLA). Minimum Linear Arrangement is a

known NP-hard problem, for which researchers have explored many heuristics [62]. Because access graph edges are weighted, we use the weighted variant of the problem.

Definition 3.3.1 (Weighted Minimum Linear Arrangement) *Given a graph*
$G(V, E)$, $|V| = n$,
find a one-to-one function
$\varphi : V \to \{1, .., n\}$
that minimizes the Linear Arrangement cost (LA), defined as
$LA(G, \varphi) = \sum_{(u,v)\in E} |(\varphi(u) - \varphi(v)) * w|$

Definition 3.3.1 states that Minimum Linear Arrangement has the objective of linearly laying out graph nodes so that it minimizes the distance between connected nodes. The weighted version of the problem prioritizes reducing the distance between pairs of nodes with stronger edges. In the context of access graphs, MinLA heuristics will try and place objects that are frequently accessed contemporaneously as close as possible in the memory layout.

In our work we use the Spectral Sequencing [40] heuristic proposed by Juvan et al. to approximate solutions to MinLA on access graphs' communities.

Definition 3.3.2 (Spectral Sequencing) *Spectral Sequencing computes the Fiedler vector of the graph G – the eigenvector $x^{(2)}$ corresponding to the second lowest eigenvalue λ_2 of the Laplacian matrix L_G of the graph G.*
It then produces the ordering function φ such that
$\varphi(u) < \varphi(v) \Leftrightarrow x^{(2)}(u) < x^{(2)}(v)$

Spectral Sequencing was shown [62] to give results of good quality, at a low computational cost. Hierarchical Memory Layouts use Spectral Sequencing as a sub-algorithm, but it can be replaced with any suitable MinLA heuristic.

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

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

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

Our use of both community detection and Minimum Linear Arrangement heuristics begs the question: Why not use Spectral Sequencing to lay out the entire access graph? This is a valid question, and using only Spectral Sequencing would produce good layouts. However, a linear layout of nodes in a graph obscures a property that is needed for our data-driven spatial locality technique. Data-driven spatial locality needs *groups* of data objects to use as training class labels (the whole process is described in detail in §3.4). Community detection algorithms output groups with desirable properties – strong connections within a group, and weak connections to nodes in other groups.

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

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

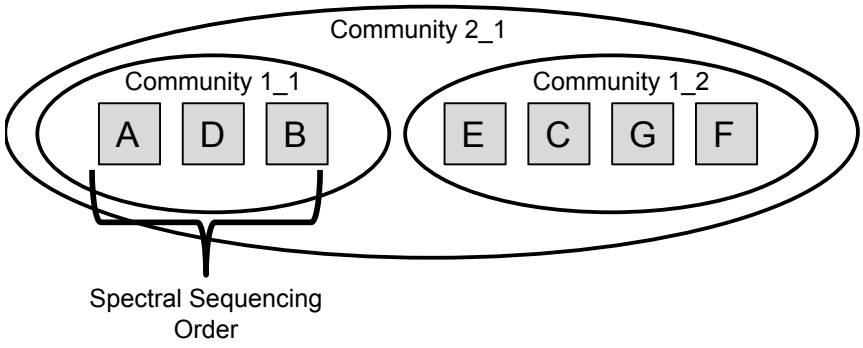


Figure 3.3: Hierarchical Memory Layout example

Figure 3.3 shows a simple example of the information produced by Hierarchical Memory Layouts. The two detected first level communities ({A, D,

B} and {E, C, G, F}) are internally ordered by Spectral Sequencing. They belong to the same second level community and will be placed next to each other in the final layout.

We use the output of Hierarchical Memory Layout for two purposes. First, we rearrange the objects in the memory access trace according to the layout and feed the new trace into the cache simulator (§3.5.1) to estimate whether improving data layout is likely to improve performance. Second, we train a machine learning model (§3.4) to find out if any data features of the objects can be used during data allocation phase to improve performance in a concrete program.

3.4 Data-driven locality

Prior work on memory layouts uses expert domain and algorithm knowledge to obtain better layouts. These solutions often use features of the data itself, such as object fields, to inform the generation of layouts. For example, in mesh traversals, the visitation order goes from one mesh object (triangle, point, edge) to its neighbours. Thus, grouping these objects in memory based on their spatial proximity (neighbours are close together in Euclidean space) yields layouts that exhibit better spatial locality. Another example are iterative graph algorithms such as PageRank. Solutions like GraphChi [42] group edges by source and destination nodes to improve spatial locality. However, the process of deriving these layouts is largely manual and relies on expert knowledge in both the algorithm's domain and memory optimization.

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

The question our technique aims to answer is the following:

Given the Object Map described in §3.2, and first level communities detected by Hierarchical Memory Layout algorithm, is it possible to decide which community an object belongs to, based only on its data features?

We use random forest classifiers [15] for this task. Random forests are a learning method for classification and regression that utilizes a combination of multiple decision trees and overcomes the decision trees' tendency to overfit. We chose random forests for two main reasons:

1. They give good results and are relatively easy to set up compared to other classifiers such as neural networks.

2. They are less opaque than other techniques. This means that it once a random forest learns to classify data from a dataset, it is possible

to extract the contribution of input vector elements to the decision process (further explained in §3.4.3).

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

3.4.1 Generating input vectors

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

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

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

The third type of data features are meta-features. These are not present in the data itself as object fields, but rather describe some inherent properties of objects. Examples of meta-features are array indices of objects, memory addresses assigned to objects within the observed execution trace, size/type of object, allocation point in the source code, etc. A correlation between spatial locality and array index (or memory address) of objects can indicate that the current layout already does well in terms of spatial locality. A correlation between size and spatial locality would mean that one should use an allocator that bins objects based on allocation size (a common strategy [46] [29]).

3.4.2 Coverage

Our technique focuses on the relationship between contents of data that is being accessed and the sequence of accesses to it. From that point of view, in terms of coverage, it is important to capture two properties that reflect the subsequent run we are optimizing for:

- The taken code paths should exhibit the same access patterns over large data structures as the program we are optimizing for. For example, if one is trying to improve spatial locality of a PageRank implementation, they should not profile an execution of depth-first search. Even though the data structure may be the same, the access pattern is not, so the extracted relationships between features of data and access pattern would not carry over. If a program does exhibit different traversal patterns in a single run, our technique would arrive at the "least common denominator" for them. The edges contained in the access graph would be weighted as the sum of the weights of the different traversal patterns, so our technique would take both into account when producing layouts and grouping criteria.

- The data structure contents should be similar to those encountered in the target program. For example, to extract relationships between data and access patterns in a PageRank algorithm, one should make sure that the input graph in the profiling run has similar characteristics to realistic graphs (e.g. similar degree distribution). If the user fails to provide a similar input, the conclusions about the best grouping criteria might not be optimal for executions on different inputs.

3.4.3 Training methodology and evaluation criteria

Our full dataset consists of all the objects in the access graph, with their input feature vectors and community labels. When training the classifier, we generate 5-fold cross validation datasets with an 80% / 20% split between training and testing data. The 80% partition is used for training, and we verify and compute accuracy on the remaining 20%.

To extract the features that contribute the most to the accuracy (feature importance), we use the *gini method* proposed by Breiman [16], implemented in scikit-learn's [59] `RandomForestClassifier` class. This method evaluates a feature's importance as the measure of all decision tree splits that include the said feature, normalized over the entire forest. The more decision tree splits a feature is involved in, the more important it is deemed for the classifier. Categorical accuracy is the percentage of samples in the test dataset for which the classifier predicted the correct label. Top-5 categorical accuracy is the percentage of samples in the test dataset for which the correct label was within the top 5 choices of the classifier. We report categorical accuracy, top-5 categorical accuracy and feature importance information in §3.5.3.

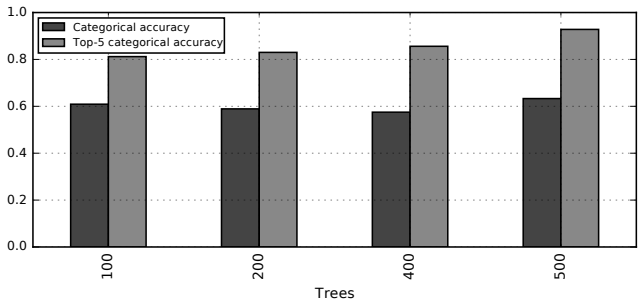


Figure 3.4: Hyperparameter search in mesh traversal

We performed all our experiments with $\sqrt{Nfeatures}$ random features considered when splitting a node. The number of trees is chosen between 100, 200, 400 and 500 trees. We performed each classifier training run on all four tree counts and found that, in general, higher tree counts yield slightly better classifiers for our datasets. In further text, all classifier results represent the best classifier we found through exploration for the given dataset. Figure 3.4 shows the categorical and top-5 categorical accuracy for our mesh dataset for different tree counts. As we can see, the categorical accuracy shows very little variance, but our classifier is able to improve the quality of mislabeled results (higher ranked correct results) with more trees. More advanced hyperparamater tuning techniques exist and can be used with our technique, however, this is out of scope of our work as it pertains to the specifics of chosen classification algorithms.

We verify that the detected fields are indeed correlated to the locality groups of objects in two ways:

1. In algorithms for which there is published research on layout optimization using data features, we verify that the properties detected by Data-driven Spatial Locality techniques have been used in prior work to inform layout design.

2. In all algorithms, we manually inspect the code under analysis. We identify the main accessor code portions for the given data structure, and reason about the sequence in which data structure is traversed and how it relates to the object properties, as visible from the source code. We show code snippets responsible for the algorithms' access patterns, and present our reasoning to support our conclusions. We perform the steps that an expert would perform to find correlations

between access patterns and data features. We then verify that the features discovered by our technique are the same as those identified manually.

3.4.4 Tidy: a memory allocator wrapper

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

The idea of hint-based allocators is not new. Chilimbi et al. [19] propose `ccmalloc` – a cache conscious allocator that accepts hints. Hints in `ccmalloc` are addresses of previously allocated objects; the allocator attempts to place the new item as close as possible to the one whose address was provided as the hint. Tidy can be adapted to use different kinds of data for hints, and we consider it a generalization of the `ccmalloc`'s approach.

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

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

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

a call to Tidy looks like:

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

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

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

3.5 Evaluation

In this section we first show how Hierarchical Memory Layouts (§3.3) can be used to estimate potential performance improvements from improved data layouts (§3.5.1). Following that, in §3.5.2 we show how HML can be used directly to derive better data layouts in storage. We show that data-driven layout techniques (§3.4) can be used to detect correlations between data features of objects and their layout, where such correlations exist, and guide dynamic memory allocations. Finally, we apply the data-driven spatial locality techniques to a subset of the SPEC benchmark suite and discuss our findings in §3.5.4.

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

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

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

Mesh traversal operates on a 2D network of node and triangle objects. Triangles contain pointers to their three nodes, and three adjacent triangles. Nodes contain their x and y coordinates and a vector of pointers to all

adjacent triangles. The data is initialized by allocating objects one by one, according to input from a node and triangle list file.

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

SPEC CPU2017 memory intensive benchmarks. SPEC CPU2017 is the 2017 release of the popular benchmark suite. For our Hierarchical Memory Layout experiments, we used three memory-intensive benchmarks (according to Amaral et al.[8]): 505.mcf, 520.omnetpp and 531.deepsjeng. 505.mcf is a mass transportation route planning program written in C. 520.omnetpp is a discrete event simulator of a large 10 gigabit network, written in C++. 531.deepsjeng is a speed-chess program with deep positional understanding, written in C++. Each manipulates a large number of heap objects, making them suitable for applying our Hierarchical Memory Layout algorithm.

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

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

We obtain the memory access traces required for generating the access graphs using DINAMITE [55]. DINAMITE is an LLVM pass that instruments every memory access and compiles the program, such that information about memory allocations, accesses, and data types is emitted to a log file. Our techniques would work with any memory access tracing tool that provides this information.

To estimate the performance effect of changing the data layout via HML, we simulate a cache hierarchy using Dinero IV[35]. The simulated Dinero cache is modelled after Intel® Core™ i5-7600K. L1 D-cache has the capacity of 64kB, and is 8-way set associative. L2 has the capacity of 256kB, and is 4-way set associative. LLC has the capacity of 8MB, and is 16-way set associative. Our DTLB simulator has 128 4kB page entries, and is 4-way associative.

3.5.1 Hierarchical Memory Layouts

The main purpose of our Hierarchical Memory Layout algorithm is to provide an *estimate of the upper bound on performance improvement from changing the data layout*. The output of the algorithm are multilevel communities produced by the first stage described in §3.3 and the final layout which maps original object addresses to addresses of the objects in the improved layout.

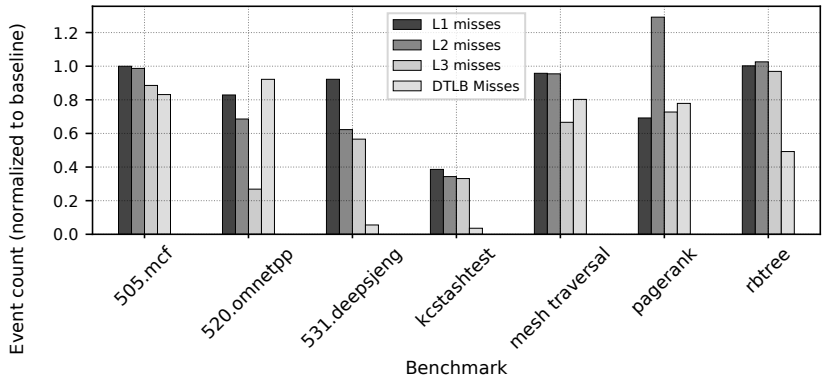


Figure 3.5: Hierarchical Memory Layouts cache and DTLB misses. Event counts are normalized to original layout results.

Figure 3.5 shows simulated cache event counts for the first seven benchmarks after applying the Hierarchical Memory Layout algorithm. The numbers are normalized to the simulated cache event counts of the original layout. Five benchmarks out of seven show significant improvements in cache miss rates.

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

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

cache line do not happen.

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

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

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

We explore two ways in which output from Hierarchical Memory Layouts can be used in practice to achieve better performance in programs. In §3.5.2 we explore the possibility of using HML to directly inform the layout of data in storage, and in §3.5.3 we show the results of applying data-driven layout techniques (§3.4) to our benchmark set.

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

3.5.2 Data layout in storage

Programs whose data layout is directly inherited from an input file, for example those that `mmap` the input file to materialize data in memory or those that dynamically allocate data objects in the same order as they appear in the input file, can directly benefit from the HML technique. We can

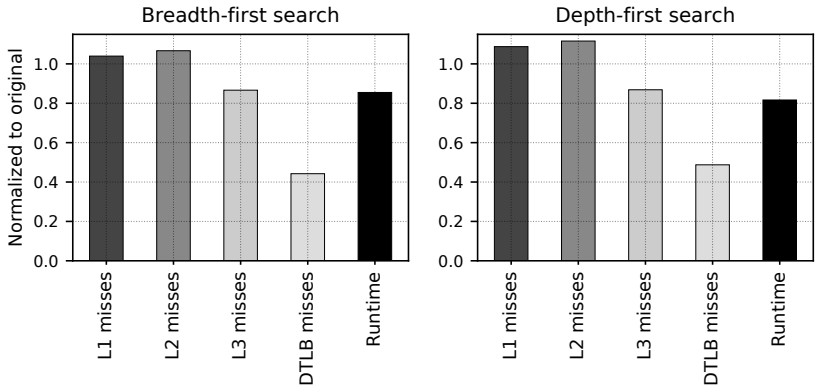


Figure 3.6: Cache misses, TLB misses, and runtime for HML-derived layouts in graph traversals. Normalized to original layout metrics.

reorganize the input data in the file in the same order as suggested by the HML algorithm and as a result obtain better spatial locality at runtime.

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

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

Figure 3.6 shows the runtime, cache misses and TLB misses obtained using the HML, relative to the baseline layout, where nodes in the input file are sorted by their numeric ID. These measurements were obtained on the actual hardware. We do not provide simulation results, because we are able to apply HML directly. We observe an improvement in runtime of 14% for the BFS and 18% for the DFS. L1 and L2 miss counts show a slight degradation, which is made up for by a significant reduction in L3 and DTLB

misses. Again, we see the pattern observed in §3.5.1, where HML tends to optimize better for L3 and TLB, while keeping L1 and L2 cache misses the same or slightly worse than the original layout. The improvements in L3 and DTLB misses outweigh this degradation and produce a positive effect on the running time.

3.5.3 Data-driven spatial locality

We applied the data-driven spatial locality techniques to the first seven benchmarks described in §3.5. In three of these, *mesh traversal*, *PageRank* and *red-black trees*, our system identified data features that could be used as hints for the Tidy memory allocator.

For PageRank and mesh traversal, our system automatically identified the same features that in the past were discovered manually: source nodes for *PageRank* [42] and Cartesian coordinates for *mesh traversal* [75]. This was a positive confirmation of the effectiveness of our techniques.

PageRank

Figure 3.7 shows that the source node is the main contributor to accurate community prediction in PageRank. The edge weight also has high predictive power, but it is not available at runtime, so we disregard it when testing new layout strategies with Tidy. Listing 3.1 shows the code that gets executed for each node `n` in the graph. The first loop iterates over all the in-edges of the node, meaning these edges would share a destination node value. The second loop iterates over all out-edges of the node, and these edges share the same source node. This means that, to optimize access locality for the first loop, one would group edges based on their destination nodes, and to optimize for the second, they would group based on the edges' source nodes. Our dataset is a snapshot of a Twitter follower graph. The graph has a power law distribution of in-degrees – few highly popular nodes with many followers and many nodes with few followers. Popular nodes will be a destination for many edges. In order to detect destination node field as a feature correlated with locality of access, edges with the same destination node would need to belong to the same community. However, the community size in our algorithm decides on community sizes based on the edge weights in an access graph, typically producing communities that are smaller in size than the in-edge counts of highly popular nodes. This means that observing the first loop in listing 3.1 for one such node, the same destination feature would be distributed over many locality communities. Due to the nature of the graph, source edge feature corresponds more closely to the structure of communities, which is what our classifier has detected

successfully.

In GraphChi [42], edges are grouped by their destination node, and edges within a group are ordered based on the their source node IDs. Grace [63] groups edges based on their source node, and sorts groups based on their destination node. Chronos [33] processes temporal graphs, and stores each epoch's edges in segments grouped by their source node. *Our technique was able to automatically detect source node as a grouping factor in our PageRank implementation – a grouping criterion used in graph processing systems optimized for data locality.*

Listing 3.1: PageRank implementation loop body

```
1   //read neighbors
2   int num_in_edges = n->num_inedges();
3   if(num_in_edges > 0){
4       for(int j=0; j<n->num_inedges(); j++){
5           sum += n->inedges[j]->weight;
6       }
7   }
8
9   //compute pagerank and store
10  float pagerank = RANDOMRESETPROB + (1 -
        RANDOMRESETPROB) * sum;
11  float cmp = fabs(n->data - pagerank);
12  if( cmp > 0.00000001){
13      value_changed = 1;
14  }
15  n->data = pagerank;
16
17  //distribute pagerank to the other nodes
18  int num_out_edges = n->num_outedges();
19  if(num_out_edges > 0){
20      float pagerankcont = pagerank /
            num_out_edges;
21      //int sum = 0;
22      for(int j=0; j<num_out_edges; j++){
23          n->outedges[j]->weight = pagerankcont;
24      }
25  }
```

Mesh traversal

Figure 3.8 shows importance scores of mesh node fields in the mesh traversal algorithm. We can see that the x and y Cartesian coordinates were picked up by the random forest as being the most important for classification. Listing 3.2 shows the main traversal loop in our implementation. The

algorithm runs for a predetermined number of steps, starting from the first triangle in the mesh. Traversal follows the path through least visited neighbouring triangles, and for each triangle touches all of the triangles points by reading and accumulating the points' coordinate values. Two triangles that get processed in succession will thus have similar spatial coordinates, and so will their points. In prior work, mesh layouts typically follow similar strategies of placing geometrically close points and triangles close together in memory.

Cache oblivious mesh layouts [75] order vertices to minimize the layout distance between neighbors. This approach effectively groups vertices that are close to each other in the mesh – their coordinates have similar values. Other layout algorithms [9] [71] [68] place neighbouring objects in volumetric grid cells that are laid out in memory according to space-filling curves. Space-filling curves linearize multi-dimensional spaces for improved spatial locality. Volumetric grid cells that are close together in domain space (have similar coordinates) will be placed together in the layout. Gopi et al. [31] produce mesh layouts by organizing objects into single-triangle strips. A single-triangle strip is a linearization technique that follows a path of neighbouring triangles until encompasses the entire mesh. Again, this technique groups neighbouring triangles in the layout – ones with similar spatial coordinates.

Our analysis of our program's access pattern and literature review support our framework's output – that x and y fields (spatial coordinates within the mesh) correspond to locality groups.

The community prediction accuracy is high, meaning we can use the x and y coordinates to group objects at allocation time with Tidy.

Listing 3.2: Mesh traversal main loop

```
1  while (steps-- > 0) {
2      int idx = find_min_neighbour(triangle);
3
4      next = triangle->neighbours[idx];
5
6      triangle->data += 1;
7      triangle = next;
8
9      touch_points(triangle->points[0], triangle->
           points[1], triangle->points[2]);
10 }
```

Red-black trees

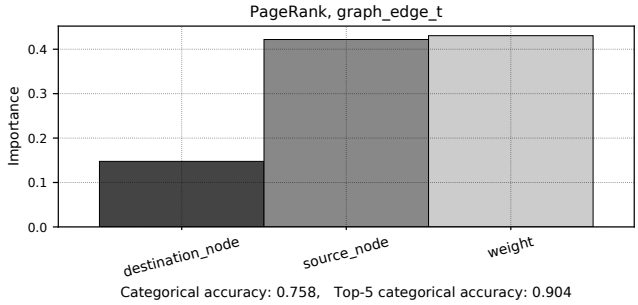


Figure 3.7: Data feature importance and categorical accuracies, PageRank

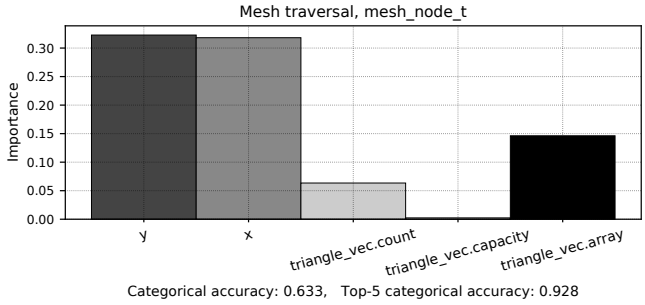


Figure 3.8: Data feature importance and categorical accuracies, mesh traversal

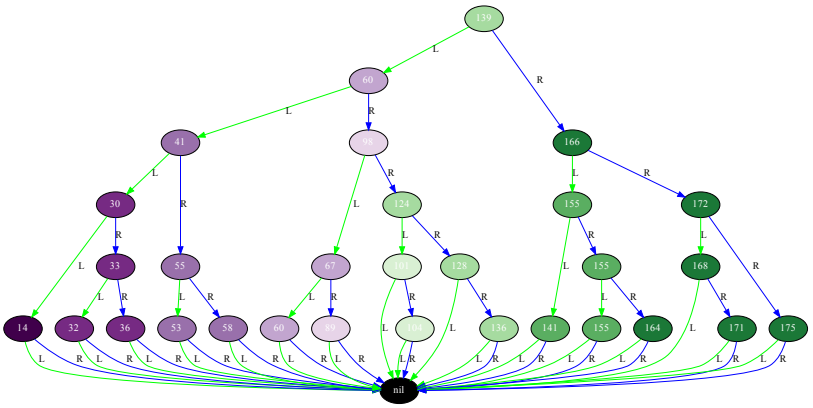


Figure 3.9: Red-black tree grouping example

Red-black trees are considered difficult to optimize for spatial locality, and we are not aware of any heuristics used in the past to improve their layout. Our system, on the other hand, was able to discover one. Figure 3.14 shows that the payload field, which is used to rebalance the tree, has the highest predictive power. Listing 3.3 shows the code of our red-black tree search implementation. Intuitively, small values will be stored in the left side of the tree, and large values in the right side. To illustrate how this reflects on grouping objects by payload, we refer to figure 3.9. The figure shows a small red-black tree with 30 nodes, containing random payloads in the $(0, 200)$ range. Leaf nodes connect to the sentinel node, which is a null value guard object in our implementation. We use this example tree to verify that the conclusion of our analysis – that payload corresponds to access locality – is indeed true. We assigned different colours to nodes, based on 20-wide payload binning. From this example, we can see that grouping objects by value ranges of their payload has the property of splitting the tree into vertical slices. A single tree search execution will visit nodes along a path from root to one of the leaves in the worst case. From the figure, we can see that such paths are localized to few colour groups. A search path corresponds to access pattern locality, and colour groups in figure 3.9 correspond to payload value similarity. From this, we conclude that our algorithm did find a meaningful correlation between locality groups and the payload feature value.

Listing 3.3: Red-black tree search function

```
1     rbt_node_t *cur = T->root;
2     while ((cur->payload != value) && (cur != T->
          sentinel)) {
3         if (value < cur->payload) {
4             cur = cur->left;
5         } else {
6             cur = cur->right;
7         }
8     }
9
10    if (cur == T->sentinel) {
11        return NULL;
12    } else {
13        return cur;
14    }
```

With categorical prediction accuracy of 0.98, red-black trees have the features with the strongest predictive power of all three benchmarks.

Hint-based allocation

We used the discovered fields in all three benchmarks to generate hints for Tidy allocator wrapper. Our mapping is relatively simple, performing one or two-dimensional binning. We bin values into buckets by dividing the value space into a grid. Each grid cell corresponds to one integer hint value. The grid size was experimentally tuned to the best performing value, and the performance numbers on real hardware are reported in Figure 3.15.

The results of our Tidy experiments are in line with the results from the simulated evaluation presented in §3.5.1. We see a runtime improvement of 25% for mesh traversal, 27% for PageRank and 14% for red-black tree queries. These improvements correlate with the reduction in cache/DTLB misses. We observe the same trend we saw in simulation – grouping items with Tidy does better for memories with higher latencies, in case of red-black trees even degrading L1 and L2 miss counts by 20%. As we observed earlier, HML is better at macro grouping than at micro grouping, providing improvements at the higher level of the memory hierarchy (L3 and TLB), but not necessarily at the lower level. This will sometimes result in L1 and L2 cache miss degradation, but L3 and DTLB improvements typically outweigh these losses.

Discrepancies in numbers between the simulation and Tidy results can be attributed to two factors. First, Tidy is a best-effort layout strategy. It opportunistically allocates objects within the same memory buckets without ordering them in the exact same way as HML would. This is a limitation of

using dynamic allocation – we cannot expect the program to anticipate the exact place in memory where each object should be stored. Second factor is the imprecision in the simulator itself. We use Dinero IV, configured to mimic the caches of our test hardware, however, we have no way of knowing how it differs from the hardware itself.

To show that the layout obtained with Tidy is responsible for the runtime improvements we observed, we show detailed stall measurements for the three benchmarks in figure 3.10. Instruction cache miss rates for each benchmark fall under 1%, meaning that interference between code and data caching is insignificant, so we omit instruction cache reports for brevity. For each benchmark, we measure only the data structure traversal segments (excluding setup and allocation). Each benchmark executes the same amount of instructions over the course of our measurements. Further, the code that is being measured does not differ between baseline and Tidy layout versions. Figure 3.10 shows a significant reduction in memory stalls over all levels of hierarchy. In the controlled context of our experiments, we can conclude that the layout imposed by Tidy did indeed reduce the amount of time CPU spends stalled during execution.

Finally, figures 3.11, 3.12 and 3.13 show memory performance of the three analyzed benchmarks, obtained by changing the bucket size from 64B (cache line size) to 4096B (page size) in a geometric sequence. The results indicate that increasing the bucket size improves performance in most cases. Smaller buckets will fill up faster, thus requiring Tidy to allocate new ones. With a random allocation sequence, this means that new buckets associated with the same hint will be scattered around in memory.

In the red-black tree benchmark, we see a degradation of L1 and L2 performance with the increase in bucket size, while LLC and DTLB misses improve. The size of a red-black tree node in our experiment is 40 bytes. If we group nodes in buckets of 64 bytes, we can fit only two nodes per bucket. As the buckets are allocated in a random sequence, the final layout will resemble the `malloc` baseline more than the ideal grouping obtained by HML. As shown in previous experiments, HML does worse in L1 and L2 misses for red-black trees and we observe a gradual increase in miss counts on these levels as we move from a layout resembling baseline to a layout resembling simulated HML results.

3.5.4 Benchmark suite experiments

In this section, we show our experiments with running the full data-driven spatial locality pipeline on a set of benchmarks. This work is done towards

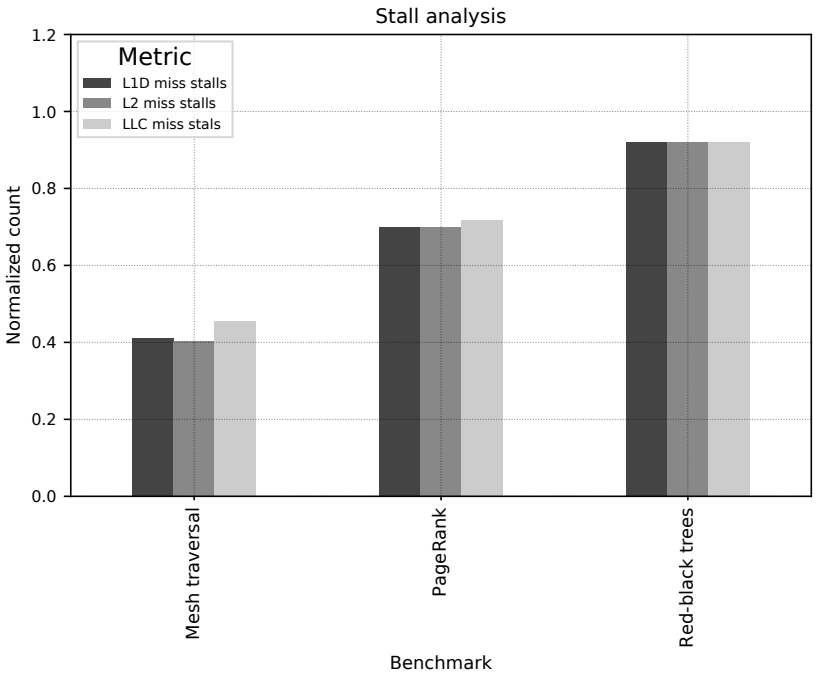


Figure 3.10: Stall breakdown

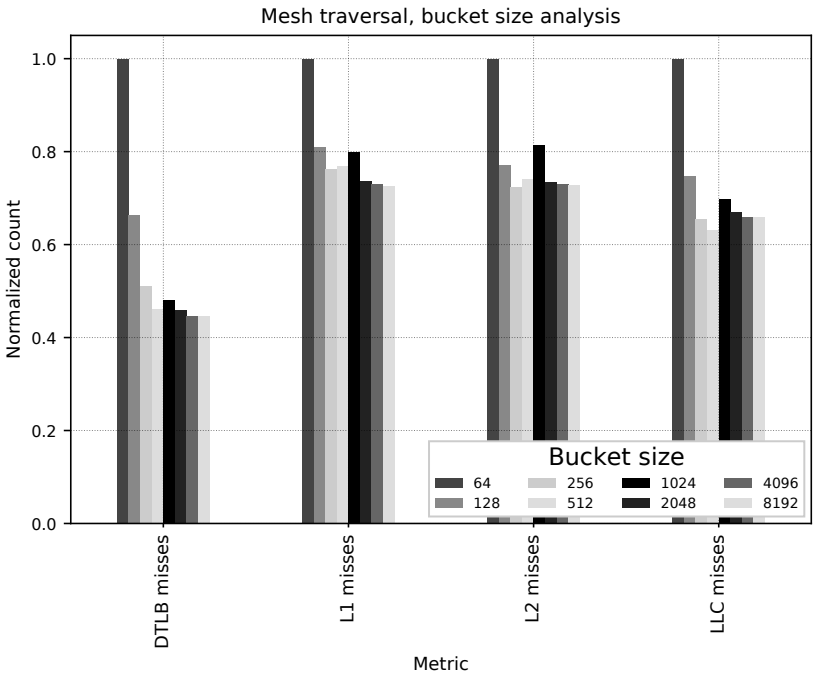


Figure 3.11: Mesh traversal – Memory performance comparison between different bucket sizes

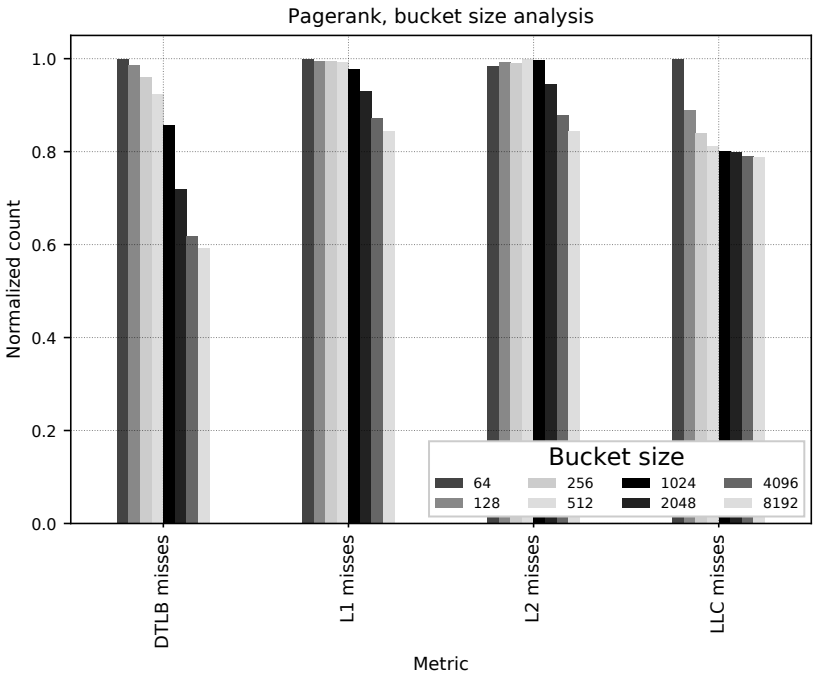


Figure 3.12: Pagerank – Memory performance comparison between different bucket sizes

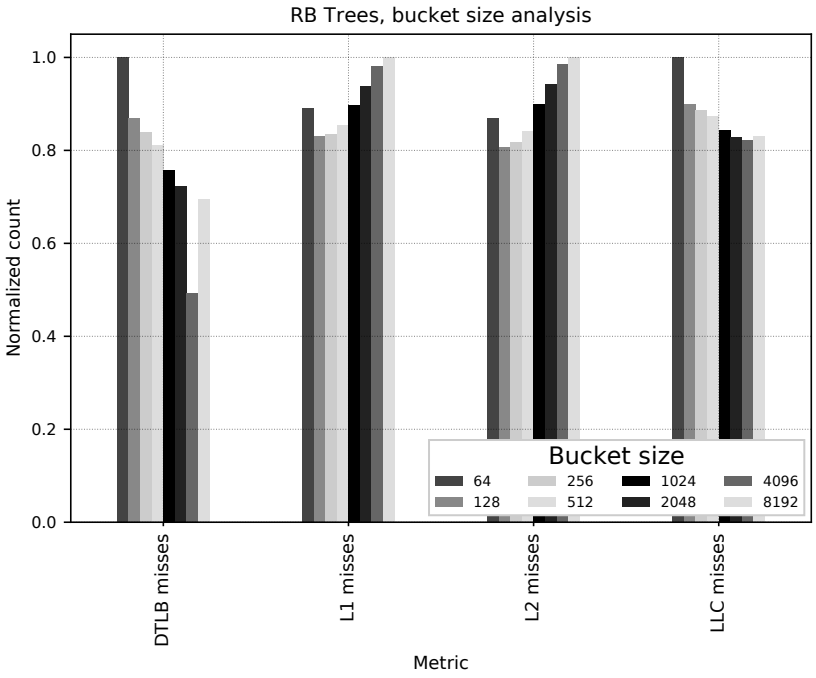


Figure 3.13: RB Trees – Memory performance comparison between different bucket sizes

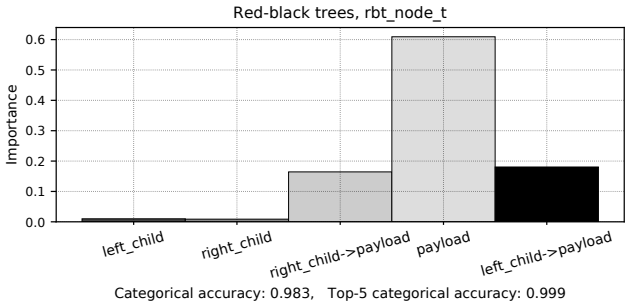


Figure 3.14: Data feature importance and categorical accuracies, red-black trees

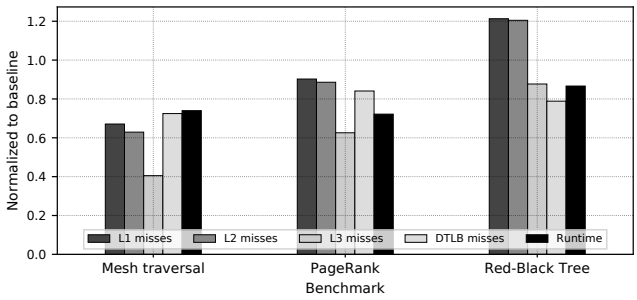


Figure 3.15: Performance improvement from using Tidy allocator wrapper with hints based on the knowledge extracted by random forests

the validation of our hypothesis 1.2.

Selecting benchmarks

For our evaluation, we chose the most recent versions of two popular benchmark suites - SPEC CPU2017 and PARSEC 3.0 apps suite. SPEC and PARSEC benchmarks are standardized versions of various scientific and industrial algorithms that have been designed and optimized by experts. The benchmarks were compiled with `gcc` and run with `perf` to collect hardware performance counters. Due to problems with the 2017 version of SPEC benchmarks, we omit cache measurements for 520.omnetpp (missing data files), 500.perlbench (crash), 502.gcc (compilation fails), 503.bwaves (does not finish). The 500.perlbench benchmark crashes near the end of execution, so we did perform the data-driven spatial locality analysis on it, up to the point of crash. We considered the remaining 21 benchmarks from SPEC, and 5 benchmarks from PARSEC.

Our final selection was based on two criteria: *the benchmarks need to be written in either C or C++* and *they need to put significant strain on the memory hierarchy*. The first condition is due to the fact that our tracing framework currently supports C and C++, which eliminated a number of SPEC benchmarks written in Fortran. The second condition is imposed by our goal of improving memory performance.

We measure the memory performance of programs by collecting hardware performance counter values with *perf*. The selected performance counters are *percentage of time the program was stalled waiting on any memory operation*, *cache miss rates for L1, L2 and LLC caches*. The measured per-

formance is shown in figure 3.16. We set a cut-off point of 30% for LLC misses and time spent waiting on memory to capture all the significant high values of these metrics across the benchmark suite. The cut-off is represented by a dashed horizontal line in figure 3.16. Table 3.1 shows a list of all the benchmarks we considered, along with whether they were included in our data-driven spatial locality analysis, and a short summary of our findings. We colour-code the table entries for readability – red entries are the ones that were not considered due to some limitation (typically compiler incompatibilities, or build/runtime problems), blue entries are the benchmarks which we do not consider memory intensive (according to previously discussed criteria), and black coloured entries are the ones that we performed data-driven spatial locality analysis on. In the latter group of benchmarks, bold typeface shows the ones we discuss in more detail in further text.

The general steps of our data-driven spatial locality analysis are outlined in chapter 3.4. For the selected suite of benchmarks, first we generate an access graph and an object map by observing all the allocations the program makes. We extract the access graph communities, and form feature vectors for each detected community, across all community levels. We split the input data – feature vector and label pairs – into training and test datasets with an 80%/20% ratio. We then feed the training feature vectors, paired with community labels into a random forest classifier, and observe errors from the subsequent classification of the test dataset. We always pick the community level that yields the lowest classification error. If this first analysis pass does not succeed in learning the classification, we re-generate the access graph and object map using only the three data structures with the highest allocation counts within a program and repeat the process. We omit primitive types from our analysis, as they are always allocated as parts of arrays.

Out of the benchmarks selected for analysis, *557.xz* uses most of its allocated memory on opaque buffers – these we have no programmatic way of analyzing with current tools, *523.xalancbmk* and *519.lbm* and *fluidanimate* reveal no correlation between accesses and data.

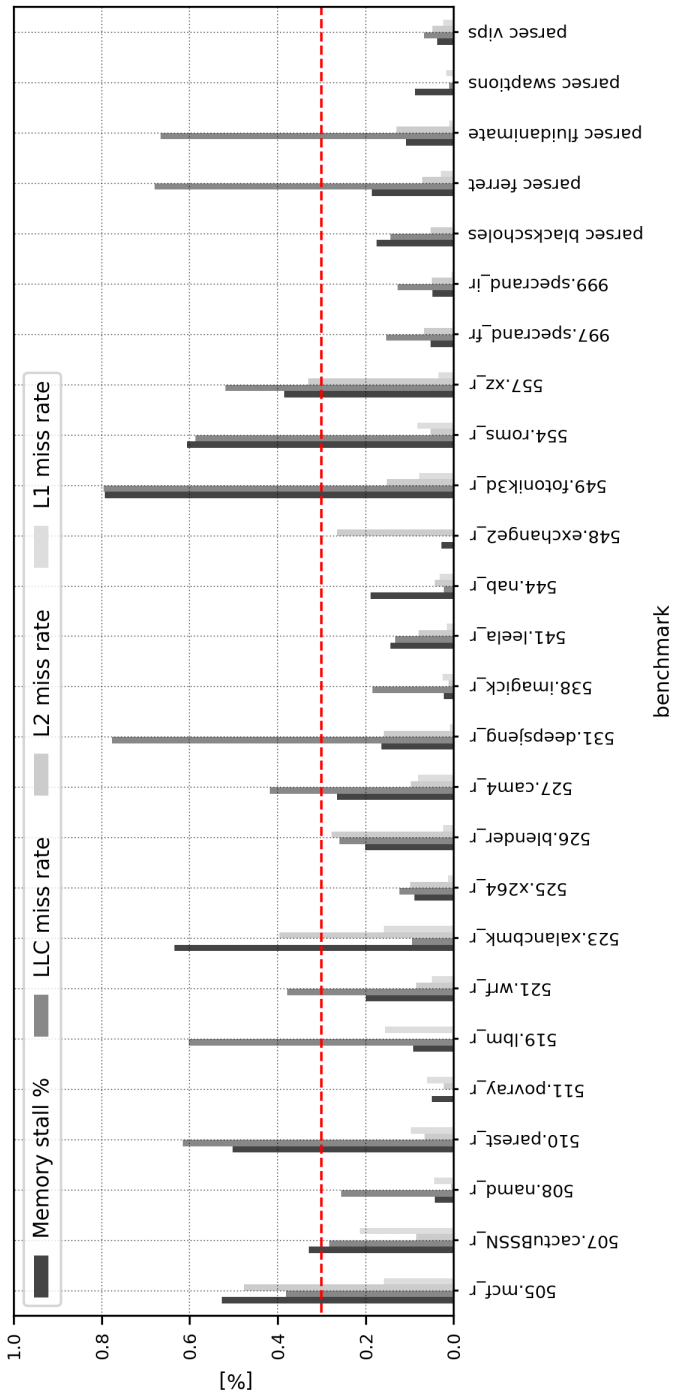


Figure 3.16: Performance measurements for SPEC CPU2017 and PARSEC 3.0

Results

505.mcf

No correlation between access patterns and data was detected in the full trace of *505.mcf*. However, when we filtered the access trace so that it contains only the accesses to the most allocated data structures, `arc` and `node`, our classifier detected correlation between the `arc` objects' positions in the allocated array and the access pattern of the algorithm. Both `arc` and `node` are allocated in large arrays, and the `arc` data structure is accessed linearly in the function `primal_bea_mpp()`. This property has been successfully detected by our tools, and there is no optimization we can apply in terms of laying out separate objects since the accesses to array elements are already linear. However, in our prior analysis of the CPU2006 version of the same benchmark, discussed in section 2.3 , we discovered that *mcf* iterates over a subset of data fields in an Array of Structures (AoS). We have shown that both extracting cold fields in a separate data structure (§2.3.2), and converting the data into the Structure of Arrays (SoA) (§2.3.1) format improves performance.

510.parest First, we analyzed *510.parest* by looking at the access graph created from allocated objects of all types. Our tools have found no correlation between data fields and the access graph communities. We then focused our analysis on the two most-allocated data types: STL's `_Rb_tree_node` and the `Point` class from *510.parest*'s `dealii` namespace. With a categorical accuracy score of 72.06% and top 5 categorical accuracy of 92.06% for RB tree node classification, our predictor can recognize a correlation between the data fields and access graph communities. The random forest classifier shows that the strongest candidate feature of RB nodes is the allocation sequence number (ordered integers increasing with each allocation made during execution) followed by the addresses of parent and child nodes. This indicates that the memory placement of nodes is already aligned with the locality of access.

531.deepsjeng

This benchmark is based on a commercial chess engine Deep Sjeng [58]. The engine solves different chess placements using a proprietary algorithm.

Analyzing the data object clusters with our random forest classifier yields a categorical accuracy of only 42.19%, however, top 5 categorical accuracy is very high at 89.44%. Upon manual inspection, we discovered that the low categorical accuracy can be contributed to different clusters having very similar feature values.

This is due to the fact that the structure of clusters is dictated by the

Table 3.1: SPEC CPU2017 and PARSEC analysis summary

Benchmark	Included	Summary
500.perlbench	No	Crashes during execution
502.gcc	No	Compilation fails
520.omnetpp	No	Missing data files
508.namd	No	Good memory performance
511.povray	No	Good memory performance
525.x264	No	Good memory performance
526.blender	No	Good memory performance
538.imagick	No	Good memory performance
541.leela	No	Good memory performance
544.nab	No	Good memory performance
548.exchange2	No	Good memory performance
997.specrand_fr	No	Good memory performance
999.specrand_ir	No	Good memory performance
blackscholes	No	Good memory performance
swaptions	No	Good memory performance
vips	No	Good memory performance
519.lbm	Yes	No correlation found
523.xalancbmk	Yes	No correlation found
557.xz	Yes	Mostly opaque buffers
fluidanimate	Yes	No correlation found
505.mcf	**Yes**	**Found correlations between array index and access pattern for `struct arc`. This means accesses to the array are already linear.**
531.deepsjeng	**Yes**	**Found correlations for the `hash` field of `ttbucket_t`. Detected field is already used for indexing into arrays.**
510.parest	**Yes**	**Found correlation between allocation sequence and access pattern for RB tree nodes. Objects allocated together are already close together in memory.**
ferret	**Yes**	**Found correlations in a small data structure that gets moved a lot. Cannot apply hint-based allocator optimizations to objects that move around the memory space.**

access graph and the community detection algorithm. Even when the correlation between access patterns and field values exists, the size and boundaries of access graph communities determines whether this correlation will be visible to the random forest classifier. To illustrate, figure 3.17 shows three different feature/community groupings. Gray boxes depict data objects, and the inscribed numbers are feature values. Figure 3.17a shows the ideal scenario where the detected communities correspond well to the object features. Figure 3.17b shows a scenario illustrating our findings in 531.deepsjeng: similar feature values belonging to different clusters. This results in lower categorical accuracy because similar feature vectors map onto different labels, making it difficult for the classifier to make good predictions. However, in this scenario, the top-k accuracies will still be high, as all the similar communities will get high votes from the classifier. Figure 3.17c shows the opposite, where the communities are too large, and different feature vectors are mapped to a single community. We have not encountered this in practice, but it is possible because the output of community detection is based solely on access graph edges.

It is important to note that, while high categorical accuracy is desirable, low values are not indicative of absence of correlation. The user needs to observe the combination of community structure and size, categorical accuracy and top-K categorical accuracy to draw accurate conclusions.

Our random forest classifier has identified the `hash` field of `struct ttbucket_t` as being tied to locality. Source code inspection revealed the `hash` field to be used as the first index into a 2-dimensional array of `struct ttbucket_t` objects that store computation results. In both of the main accessor functions `StoreTT()` and `ProbeTT()` the array is indexed into using the calculation provided in listing 3.4.

Listing 3.4: 531.deepsjeng: Index calculation based on hash value

```
1    index = (unsigned int)nhash;
2    temp = &(TTable[index % TTSize]);
```

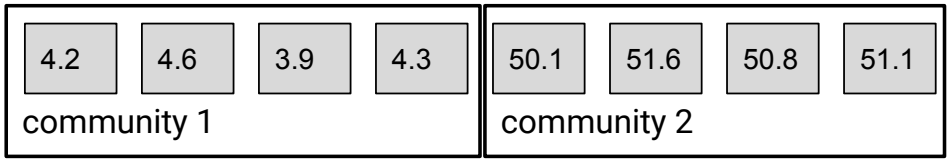
The algorithm traverses this object store (`temp` pointer in listing 3.4 linearly over its second index. As such, these objects are already grouped based on their `hash` fields and we can confirm that the identified correlation is indeed present, and proper grouping implemented.

PARSEC ferret

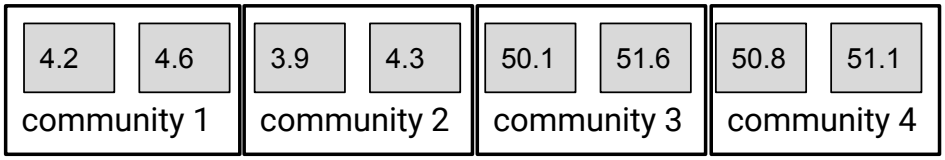
PARSEC's ferret benchmark is based on the Ferret toolkit [52] for content-based similarity search of feature-rich data such as audio recordings, digital

Figure 3.17: Examples of community groupings

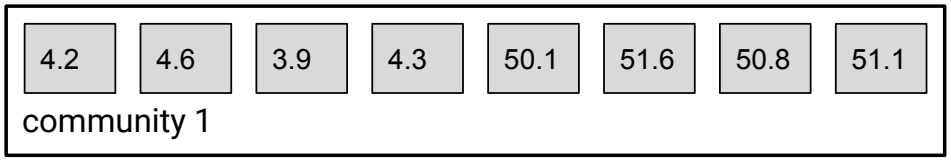
(a) good community grouping



(b) similar features mapping into different communities



(c) different features mapping into the same communities



images and sensor data. The version included in the benchmark suite is configured specifically for image search.

Our analysis yielded a classifier with categorical accuracy of 63.56%, and top-5 categorical accuracy of 78.95%. The most important features for the classifier were identified to be `reg1` and `reg2` fields in the `RegionPair` data structure. The start of the main accessor loop for the `RegionPair` data structure is shown in listing 3.5.

Listing 3.5: ferret: Sorted array accesses

```
1  sorted = bucket_sort ( pair, num_edges );
2
3  /* Merge similar regions */
4  for ( ik = 0; ik < num_edges; ik++ )
5   {
6    reg1 = find_set ( parent, sorted[ik].reg1 );
7    reg2 = find_set ( parent, sorted[ik].reg2 );
```

The algorithm, however, relies on the `RegionPair` array being sorted by a calculated cumulative histogram value (shown in listing 3.6).

Listing 3.6: ferret: Sorting criterion

```
1  /* Perform bucket sort */
2  for ( ih = 0; ih < num_elems; ih++ )
3   {
4    sorted[cum_histo[pair[ih].delta]++] = pair[ih];
5   }
```

While it is possible to add a level of indirection to the accesses (allowing us to sort pointers instead of objects), doing so would not be beneficial in this case since the data structure is only 12 bytes in size – only 50% larger than a 64-bit pointer.

Conclusion

The evaluation presented here shows that a subset of the selected benchmarks exhibits correlation between data features and the programs' access patterns. However, existing software is often written in such ways that the use of Tidy allocator is impossible without rewriting the code. The main obstacle to applying Tidy that we identified are arrays of objects. Access

semantics of arrays are index-based. This means that the algorithm will access the data objects using their integer indices. In such cases, the identified link between data features and access pattern cannot be exploited out of the box, because a new layout would alter the indexing scheme of the data set. For example, consider an implementation of a hash-table using an array of linked lists. In such a data structure, the indices are calculated using a hash function. Exploiting the discovered correlation between object features and access pattern would require writing a new hash function. Kraska et al. explore the possibility of using neural networks to learn better indexing schemes[41], and there is a potential for future work in combining their work with our Data-Driven Spatial Locality techniques.

Using arrays is commonly understood to be better for performance than using linked lists or other dynamically changing data structures. This is true as long as the access pattern of the array items is linear or strided. Our study shows that attempting to design performant data structures from the get-go often leaves out opportunities for applying Data-Driven Spatial Locality techniques at a later stage. This aligns with our observation that expert-optimized data layouts typically approach layout as a first-order citizen in the design process.

To conclude, we note three classes of programs with respect to the applicability of our Data-Driven Spatial Locality techniques:

1. **Programs in which correlations are present, and applying hint-based allocation is feasible**. This category contains programs with "textbook" versions of pointer-based data structures. Objects are allocated separately and no memory optimizations are applied in the original code.

2. **Programs in which correlations are present, and point to optimizations that are already applied**. An example of this class of programs is *531.deepsjeng* in which our techniques detected a correlation between hash values used to index into arrays, and the program's access pattern. Our techniques can be used to verify that such optimizations are already applied.

3. **Programs in which correlations are present, but changing the layout would require significant changes to the program**.

Chapter 4

Related work

Several instrumentation frameworks have been designed previously with the goal of solving memory bottleneck problems. Limitations of these frameworks include targeting a specific programming language, providing coarse grained instrumentation abstractions, or providing fine grained instrumentation abstractions that are sampled to reduce overhead. Other tools are not as flexible in that users can only view output through a user interface, or are designed for a specific use case.

Existing instrumentation frameworks like Pin [51] and Valgrind [56] instrument programs and allow programmers to build dynamic analysis tools on top of them much like DINAMITE. Pin achieves instrumentation through a highly optimized JIT compiler that intercepts the first instruction of a supplied executable and compiles instrumentation functions to execute where appropriate. Pintools are created by the programmer using the Pin framework that describe the instrumentation analysis. Runtime binary instrumentation is limited in that information gleaned from the framework does not translate to code level data structures out of the box. Without knowing what machine instructions relate to higher level data, programmers do not immediately see patterns related to their original work. DINAMITE instrumentation is inserted at compile-time allowing it to retain code level information about data that is valuable to the programmer using the tool.

Valgrind is another framework that performs runtime binary instrumentation. It addresses the lack of code level information other runtime analysis frameworks suffer from by supporting a technique called "shadow values". Shadow values replace values in memory and registers with values that describe them. Valgrind requires tool writers to implement their own shadow values – a technique difficult to implement. The framework supports the implementation of shadow values by providing registers to store shadow values and extra output functionality for printing these values during execution. Though this technique enriches instrumentation by providing more context about memory accesses during execution, it is a complex task in practice, and can easily slow down instrumentation if not done carefully. DINAMITE provides memory access information by default; each log entry contains the

address of the memory location accessed and is available for tool writers to query, though register information is absent but can be easily extended using the llvm.read_register instrinsic [2]. Programmers using DINAMITE are not left to implement complex and bug-prone functionality to obtain memory access information.

Zhao et al. [79] describe a tool designed to detect true and false sharing built on top of the memory shadowing framework Umbra [78]. Umbra is a performant implementation of shadow values while remaining general purpose, and is achieved by mapping only allocated memory instead of the whole address space. Memory sharing is detected by associating cache lines with shadow memory exposed by Umbra. Association of thread to address is done via a bitmap describing thread ids responsible for each access. Sheriff [48] addresses the same problem, but requires either the programmer to rewrite source code or rely on sampling to catch culprits, and is specifically designed to detect false sharing. DINAMITE achieves the same result by inserting thread IDs in each log entry, which also contain accessed addresses. The log entry also contains all the source level details necessary to pinpoint exactly where in the code the accesses were performed and to what data type.

Memprof [43] is a tool that profiles objects that make remote memory accesses on NUMA machines so programmers can potentially minimize them. Memory accesses are measured through instruction-based sampling (IBS) that relies on hardware support and requires using a kernel module, constraining the tool to the Linux/AMD platforms. DINAMITE can be extended with a native plugin or Spark kernel to generate the same information. It sacrifices performance over accuracy and portability as it does not rely on hardware support for instruction sampling.

Similar to Memprof, DProf [61] uses IBS to acquire memory traces to locate data types that stress the cache. Programmers can view data type statistics and how they behave in the cache, what data types generate the most cache accesses and misses, and the most common functions that operate on these data types. DProf required changes to the operating system, which is significant barrier for its adoption. The flexibility of DINAMITE enabled us to plug in a cache simulator into the framework and to generate the same information as DProf, but with greater flexibility to add new analysis and without attachment to a particular operating system or hardware.

Other tools provide more source-level detail but are slower and less flexible. Memspy [54] provides rich information on program execution, including the execution time, miss rate, and memory stall time broken down by code and data. Details are tracked by executing the application through a mem-

ory simluator and instrumention through a preprocessor, which is not as portable as LLVM. Memspy reported a 22×-58× slowdown in execution time. DINAMITE's slowdowns are competitive with modern instrumentation frameworks and provide a pluggable framework for all kinds of data analyses.

Like MACPO [64], our tool instruments data accesses at the compiler level instead of the binary level to keep source-level information. To combat overhead, MACPO limits instrumentation to "snapshots" of program execution, that are staggered in an attempt to capture complete program behaviour. Trace size is reduced by limiting instrumentation not only to "snapshots" but also to non-scalar data types. Similarily Dprof and Memprof use IBS and only output the most commonly accessed data and their cache statistics. DINAMITE instruments all memory accesses inviting unlimited flexibility of analysis at the expense of higher runtime overhead.

Cache-conscious structure layout and definition [20] [19] blazed the trail for the ideas presented here. Collocating contemporaneously accessed objects and hint-based allocators (the previously discussed `ccmalloc`) were both explored by the authors. The access graphs allow for novel analyses that help programmers *understand which data should be grouped at runtime.*

Profile-based data layout optimizations were explored by Rubin et al. [66] Their approach uses profiling techniques to identify objects that are top offenders to cache performance. This information is used to apply a series of known layout optimizations, such as structure splitting, field reordering and reordering of whole objects. Our work attempts to solve the layout problem in a more holistic fashion – by understanding the interplay between accesses to all of the objects in a data set.

The use of data structure fields to inform data layout creation was inspired by GraphChi [42]. The authors proposed a layout specifically for bulk synchronous processing on graphs; we aspired to capture the essence of these ideas, so they could be used for data structures in general. With access graphs, we aimed to remove the necessity of domain knowledge for reasoning about good layouts. We showed that our techniques reach similar conclusions about laying out edges for Pagerank.

Higher order theory of locality (HOTL) [74] sets up a mathematical framework for thinking about different locality metrics. From it, Xiang et al. develop a novel low-overhead way of locality sampling, and demonstrate the ability to predict miss rates from the acquired information. While providing an excellent basis for reasoning about locality, techniques presented in HOTL do not expose actionable data on how to improve locality in a program. Access graphs take a different approach of observing the full access trace and

providing insight into how objects relate to each other, in terms of access locality. This level of detail, while incurring large overheads compared to sampling techniques, brings new information on what can be done to data layout to improve performance.

Yoon et al. [75] use a graph representation for generating a cache-oblivious layout of the mesh, but their representation is very different from locality graphs. Edges are formed between vertices connected in a mesh; in other words this representation is specific to the mesh data structure and requires knowing which vertices are connected. Locality graphs operate on any generic memory access trace and require no knowledge of data structure specifics.

Liu et al. propose a sampling tool [49] that correlates bad memory performance with data objects: static or dynamically allocated variables. Similar approaches can be found in DProf [61] and MemProf [43], which correlate cache misses and remote memory accesses to data items. Looking at data as the first order citizen in memory performance analysis is a good approach to helping programmers better understand memory bottlenecks. Access graphs use a similar data-centric approach, but aspire to bring more actionable insight by observing the entire execution trace and extracting locality-related relationships between data objects.

Peled et al. propose a context-based cache prefetcher model [60], that detects *semantic locality* and issues prefetches based on it. Their approach is to implement a machine learning model in hardware that observes "contexts" of memory accesses during execution. A context contains information such as register contents, access and branch histories, compiler provided type information, etc. The approach used by Peled et al., and our data-driven spatial locality method are similar in that they both base their techniques on data-centric contextual information. They are on two sides of the trade-off spectrum: semantic locality is generally applicable out of the box and has insight into lower-level information, but the amount of data it can observe at any point in time is limited due to hardware constraints. Access graphs and their related tools observe a much larger body of information about the execution, but at a higher level. They do not provide performance benefits on-the-fly, but is used to derive better locality optimizations.

Previous allocator work such as dlmalloc [46] and jemalloc [29] inspired Tidy's arena-based allocation. However, these allocators rely on the information available through regular allocation calls, namely the size of the object being allocated, to group data. Our approach with Tidy was redefining what an allocation call looks like, and showing that additional information can further improve locality. We do not compare against dlmalloc or jemalloc,

as our work is mostly orthogonal – we show that informed hints can improve a layout's performance even when allocating objects of the same size throughout the program. Tidy is a proof of concept allocator wrapper, and our implementation does not focus on the allocation speed. The insights from Tidy can be used to bring hint-based allocation into state-of-the-art allocator libraries.

Chapter 5

Conclusion

In the past decades, the gap between CPU speeds and RAM bandwidth has been increasing. To make matters worse, the amounts of data modern programs process is growing rapidly as well. Hardware architects are working on tackling the problem using technologies such as novel cache prefetchers and near-memory processing units. On the software side, experts spend a lot of time and effort devising new ways of laying data out in memory to bring better spatial locality back into programs. We have observed a trend in prior work on memory layouts of gaining knowledge about the access pattern of an algorithm, relating it to the data itself, and using that data to guide memory layout design. This process has been mostly manual, and difficult enough to warrant top-tier publications. Based on these observations, we set out to investigate whether the burden of obtaining the knowledge about the relationship between access patterns and data can be delegated to machines.

The first step towards our goal is data collection. With the advent of big-data and the revival of machine learning, the opportunity arose to analyze full, detailed memory access traces. To this end, we built DINAMITE (chapter 2), an LLVM compiler pass that instruments memory accesses, allocations and function events in a program. We show that compiler-based instrumentation can be used to deliver very rich information about every access, at a fraction of the runtime cost of previously available instrumentation tools. When compared against the `pinatrace` access tracing tool from the Pin instrumentation framework DINAMITE reduced the overhead of tracing by an order of magnitude, while delivering a higher level of details about the accesses (section 2.2.3). We show how DINAMITE's modular library structure can be used to feed access traces into existing big-data processing systems, which allows for faster exploratory memory performance debugging (section 2.3.3).

The second step towards our goal is the analysis of collected memory access traces, and formulation of techniques that enable the detection of relationships between data and access patterns. We created a novel representation of access patterns called *access graphs* (section 3.2). Access graphs

capture spatial locality properties of a program. Their nodes represent data objects allocated within a program, and their edges the number of contemporaneous accesses of two objects. Based on the spatial locality principle of caches, placing objects that often get accessed contemporaneously close in memory has the potential for improving performance. To detect groups of such objects, we use a novel combination of community detection and linear graph layout algorithms to form a new heuristic – Hierarchical Memory Layouts (section 3.3). Hierarchical Memory Layouts approximate solutions to the NP-Hard problem of optimal data layout in memory. We show that our solution manages to outperform the baseline in all observed cases in terms of LLC and DTLB miss rates 3.5.1. Further, Hierarchical Memory Layouts can be used to directly generate better layouts of data in the cases where layouts in storage match layouts in memory (section 3.5.2). Finally, we developed the eponymous Data-Driven Spatial Locality technique that detects correlations between the data itself and groups of data objects that should be placed close in memory (section 3.4). Our findings indicate that many programs exhibit such correlations (section 3.5.4), *validating our primary hypothesis*. However, applying memory optimizations based on that knowledge is feasible only for a subset of programs with simple allocation patterns, which *partially validates our secondary hypothesis*.

Our work sets a foundation for different research directions in the future. Starting from DINAMITE, the computation model which most stream processing engines provide today is not well suited for tasks such as creation of access graphs. It is worth investigating what changes need to be introduced in order to support a scalable sliding window processing of memory access traces. The overheads of using Spark Streaming with DINAMITE are notably very high – over an order of magnitude higher than simple storage of traces. There is potential for improvement here that could make exploratory memory performance debugging a more attractive and feasible option to be included in engineers' tool sets. The overhead of DINAMITE itself could further be reduced by using static instead of dynamic linkage for the logging libraries. This would tie the compiled binary to a single logging variant, which is a trade-off we have not looked into in detail.

The second avenue of potential future work would be the exploration of access graphs themselves. Access graphs are a novel tool in the context of memory optimizations, and we have explored only one application of them – grouping of objects based on graph communities. More work could be done to explore possible new uses for access graphs.

The Data-Driven Spatial Locality technique presented in this dissertation uses random forests for the sake of simplicity of use and transparency of

results – random forests deliver good results without extensive tuning, and it is possible to analyze the decision process to find features that contributed the most to successful classification. With the rapid progress happening in the area of artificial neural networks, it is reasonable to expect them to attain similar characteristics to random forests. Given the amount of work that goes into optimizing the performance of neural networks and the possibility of compiling trained networks into native code, it would be interesting to look into the feasibility of directly embedding neural networks into hint-based allocators.

Bibliography

[1] Cache memory store in a processor of a data processing system, July 22 1975. US Patent 3,896,419.

[2] Llvm language reference manual, 2016.

[3] Wired tiger: making big data roar, 2016.

[4] Wiredtiger storage engine, 2016.

[5] Wt-2029 improve scalability of statistics, 2016.

[6] Andreas Abel and Jan Reineke. Reverse engineering of cache replacement policies in intel microprocessors and their evaluation. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 141–142. IEEE, 2014.

[7] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. Dbmss on a modern processor: Where does time go? In *VLDB" 99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, number DIAS-CONF-1999-001, 1999.

[8] José Nelson Amaral, Edson Borin, Dylan Ashley, Caian Benedicto, Elliot Colp, Joao Henrique Stange Hoffmam, Marcus Karpoff, Erick Ochoa, Morgan Redshaw, and Raphael Ernani Rodrigues. The alberta workloads for the spec cpu 2017 benchmark suite.

[9] Tetsuo Asano, Desh Ranjan, Thomas Roos, Emo Welzl, and Peter Widmayer. Space-filling curves and their use in the design of geometric data structures. *Theoretical Computer Science*, 181(1):3–15, 1997.

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

[11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[12] Georgios Bitzes and Andrzej Nowak. The overhead of profiling using pmu hardware counters. *CERN openlab report*, 2014.

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

[14] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, Nickolai Zeldovich, et al. An analysis of linux scalability to many cores. In *OSDI*, volume 10, 2010.

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

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

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

[18] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*. IEEE, 2003.

[19] Trishul M Chilimbi, Bob Davidson, and James R Larus. Cache-conscious structure definition. In *ACM SIGPLAN Notices*, volume 34. ACM, 1999.

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

[21] William P Churchill Jr. Memory access technique, November 1 1977. US Patent 4,056,845.

[22] Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing

and polytope models. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pages 136–146. IEEE, 2009.

[23] Arnaldo Carvalho de Melo. The new linux'perf'tools. 2010.

[24] Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static loop bound analysis of c programs based on flow analysis and abstract interpretation. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA'08. 14th IEEE International Conference on*, pages 161–166. IEEE, 2008.

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

[26] Dave Dice, Yossi Lev, and Mark Moir. Scalable statistics counters. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2013.

[27] Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11, 2007.

[28] Jan Edler and Mark D Hill. Dinero iv trace-driven uniprocessor cache simulator, 1998.

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

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

[31] M Gopi and David Eppstien. Single-strip triangulation of manifolds with arbitrary topology. In *Computer Graphics Forum*, volume 23, pages 371–379. Wiley Online Library, 2004.

[32] Kazushige Goto and Robert A Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3), 2008.

[33] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen.

Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, page 1. ACM, 2014.

[34] John L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4), 2006.

[35] Mark D Hill and J Elder. Dineroiv trace-driven uniprocessor cache simulator, 1998.

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

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

[38] Marty Itzkowitz, Brian JN Wylie, Christopher Aoki, and Nicolai Kosche. Memory profiling using hardware counters. In *Supercomputing, 2003 ACM/IEEE Conference*. IEEE, 2003.

[39] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM, 2010.

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

[41] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

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

[43] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. Memprof: A memory profiler for numa multicore systems. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012.

[44] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and*

Optimization, 2004. CGO 2004. International Symposium on. IEEE, 2004.

[45] Chris Lattner and Vikram Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *ACM SIGPLAN Notices*, volume 40. ACM, 2005.

[46] Doug Lea. Dlmalloc, 2010.

[47] Han Bok Lee and Benjamin G Zorn. Bit: A tool for instrumenting java bytecodes. In *USENIX Symposium on Internet technologies and Systems*, 1997.

[48] Tongping Liu and Emery D Berger. Sheriff: precise detection and automatic mitigation of false sharing. *ACM SIGPLAN Notices*, 46(10), 2011.

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

[50] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. Concurrent data structures for near-memory computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–245. ACM, 2017.

[51] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40. ACM, 2005.

[52] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Ferret: a toolkit for content-based similarity search of feature-rich data. *ACM SIGOPS Operating Systems Review*, 40(4):317–330, 2006.

[53] Rama Kishan Malladi. Using intel® vtune™ performance analyzer events/ratios & optimizing applications. *http:/software. intel. com*, 2009.

[54] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. Memspy: Analyzing memory system bottlenecks in programs. In *ACM SIGMETRICS Performance Evaluation Review*, volume 20. ACM, 1992.

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

[56] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42. ACM, 2007.

[57] Preeti Ranjan Panda, Hiroshi Nakamura, Nikil D Dutt, and Alexandru Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE transactions on computers*, 48(2):142–149, 1999.

[58] Gian-Carlo Pascutto. Sjeng 11.2 deep sjeng, 2018.

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

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

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

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

[63] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. Managing large graphs on multi-cores with graph awareness. 2012.

[64] Ashay Rane and James Browne. Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012.

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

[66] Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *ACM SIGPLAN Notices*, volume 37, pages 140–153. ACM, 2002.

[67] Pedro V Sander, Diego Nehab, and Joshua Barczak. Fast triangle reordering for vertex locality and reduced overdraw. In *ACM Transactions on Graphics (TOG)*, volume 26, page 89. ACM, 2007.

[68] Shankar Prasad Sastry. Dynamic meshing techniques for quality improvement, untangling, and warping. 2012.

[69] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.

[70] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[71] Huy T Vo, Claudio T Silva, Luiz F Scheidegger, and Valerio Pascucci. Simple and efficient mesh layout with space-filling curves. *Journal of Graphics Tools*, 16(1):25–39, 2012.

[72] Michael Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664. ACM, 1989.

[73] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[74] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. Hotl: a higher order theory of locality. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 343–356. ACM, 2013.

[75] Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha. Cache-oblivious mesh layouts. In *ACM Transactions on Graphics (TOG)*, volume 24. ACM, 2005.

[76] Sung-Eui Yoon and Dinesh Manocha. Cache-efficient layouts of bounding volume hierarchies. In *Computer Graphics Forum*, volume 25, pages 507–516. Wiley Online Library, 2006.

[77] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the*, 2012.

[78] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010.

[79] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *ACM SIGPLAN Notices*, volume 46. ACM, 2011.

Appendix A

Hardware

All experiments in chapter 2 evaluation were performed on one of the following machines:

- *Machine A* - AMD Opteron 6272, 4 chips with 16 cores and 16MB of last level cache each, and 512GB of RAM
- *Machine B* - AMD Opteron 2435, 2 chips with 6 cores 6MB of last level cache each, and 32GB of RAM