Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Cache-efficient C++ for High Frequency Trading

Author:
Alexander Brady

Supervisor:
Dr Paul A. Bilokon

Submitted in partial fulfillment of the requirements for the MSc degree in Computing Science of Imperial College London

August 2022

Abstract

In the context of High Frequency Trading, the purpose of writing cache-efficient code is to achieve low execution latencies and a low standard deviation among execution latencies. However, the literature lacks robust and detailed analysis of the ways in which C++ code can use data- and instruction-caches (in)efficiently. This report aims to make progress towards filling this gap in the literature by undertaking an investigation into the cache-efficiency factors underlying latency patterns observed in the execution of C++ code. This investigation focusses on (1) cache-structure *per se* and how it imposes requirements on C++ programmers writing for low-latency applications, (2) how the cache-(in)efficiency of C++ containers can affect execution latencies, and (3) how data-member ordering and packing in user-defined C++ types affects execution latencies.

Following on from this research contribution, the report proceeds to detail the development of a novel piece of software which detects data- and instruction-cache inefficiencies on the basis of user-provided source-code files and compiled binary files. This software is highly configurable and can be tailored to specific use-cases, but, ordinarily, (1) analyses the CPU caches on which the code will be run, (2) analyses user-provided source-code files and compiled binary files, (3) alerts the user to data- and instruction-cache inefficiencies, and (4) suggests specific cache- and code-tailored mitigations to these inefficiencies.

Finally, the report concludes with an evaluation of the foregoing and some suggestions for future research directions.

Acknowledgments

Many thanks to my supervisor Dr Paul A. Bilokon and to Professor Emil Lupu for their help with this project.

Contents

1 Introduction **2**
- 1.1 Aims and approach . . . 2
- 1.2 Context . . . 2
- 1.3 Report outline . . . 3
- 1.4 Summary of achievements . . . 4

2 Background **5**
- 2.1 High Frequency Trading . . . 5
- 2.2 C++ . . . 6
- 2.3 Caches . . . 8
- 2.4 Existing software . . . 10

3 Research contribution **12**
- 3.1 Outline . . . 12
- 3.2 Cache structure effects . . . 12
- 3.3 C++ containers . . . 18
- 3.4 Data layouts . . . 29
- 3.5 Summary of conclusions . . . 43

4 Software contribution **45**
- 4.1 Outline and general use . . . 45
- 4.2 Programme setup . . . 46
- 4.3 Cache analysis . . . 47
- 4.4 Data-efficiency analysis . . . 51
- 4.5 Code-efficiency analysis . . . 56
- 4.6 Other features . . . 60

5 Evaluation **61**
- 5.1 Overall approach . . . 61
- 5.2 Cache analysis . . . 62
- 5.3 Data analysis . . . 63
- 5.4 Code analysis . . . 66
- 5.5 Robustness and usability . . . 72

6 Conclusions and future directions **74**

Chapter 1

Introduction

1.1 Aims and approach

The overarching goal of this project is to enable programmers to optimise the cache-efficiency of their C++ code in order to achieve low execution latencies. A significant barrier to achieving this aim is that many cache-related inefficiencies are opaque to C++ programmers because they arise from compilation and execution specifics such as byte-for-byte data layouts and the addresses and sizes of functions and types in main memory at execution-time.

The first step toward achieving this aim is to undertake a detailed empirical investigation into the low-level execution details that underlie cache-inefficiencies for the purpose of identifying, describing, and explaining how higher-level C++ code can be cache (in)efficient. The second step is the development of a novel piece of software that builds on these empirical findings by analysing CPU caches, source-code files, and compiled binary files in order to identify inefficiencies, flag them up to the programmer, and suggest solutions.

1.2 Context

Existing research into optimising the cache-efficiency of C++ code in order to achieve latency improvements on the order of micro- and nano-seconds is thin on the ground. This is a result of the fact that the main progenitors of cutting-edge research of this kind are High Frequency Trading companies that rely on the secrecy of their innovations for profitability and survival as successful businesses.

That said, there are some well-known books on HFT. For example, Aldridge's book attempts to bridge a perceived 'academic void' between the computational research and the financial research into algorithmic trading systems and strategies [1]; [2] focusses on the mathematical modelling that underlies algorithmic trading systems. There are also a number of publicly available conference talks given by people working in the industry on the design of HFT systems [3] and HFT in general (e.g., [4, 5, 6, 7]), but these usually cover a lot of ground in little detail, sometimes with reference to cache-efficiency but only in fairly basic and vague terms.

On the other hand, as the primary tool of software-based HFT, literature on C++ is relatively easy to come by. For example, the design choices behind C++ (cf., e.g., [8, 9]) and its widely used Standard Template Library (e.g., [10, 11]) are well documented.

Foundational research into cache design is also relevant. There are a number of widely cited publications on data- and instruction-cache design and how it affects cache-miss ratios (e.g., [12, 13]), and some further research attempts to translate this research into practical efficiency improvements (e.g., [14, 15, 16]).

As mentioned above, however, academic research into the cache-efficiency of C++ for ultra-low-latency software is virtually nonexistent. There are a number of blogs and blog posts which describe simple experiments looking at the effects of cache efficiency on execution latencies, but these only provide averaged latency data, neglecting to investigate the factors behind execution latencies (cache-access behaviour, instruction-execution behaviour, and so on) and also neglecting to provide standard deviation data ([17, 18, 19, 20, 21]). I will discuss this literature and engage with it critically in more detail in Chapter 2.

Not only is existing research of this kind hard to come by, software available for identifying and mitigating latency problems arising from data- and instruction-cache inefficiencies is limited and not enormously useful to C++ programmers. Some software exists that provides the capability to perform these analyses, but using these existing tools to identify cache inefficiencies in C++ code — and, more importantly, to explain and mitigate them — is tedious and error-prone. Examples include gdb [22], objdump [23, 24], nm [25, 26], perf [27], and Valgrind's Cachegrind functionality [28, 29]. There is also some software which automatically (i.e., opaquely) performs specific cache-efficiency optimisations at compile-time (e.g., [14, 15]) or at run-time (e.g., [16]). I will discuss both kinds of software further in the following chapter.

1.3 Report outline

Following on from the foregoing, Chapter 2 elaborates on the existing literature and software. Section 2.1 looks at HFT in more detail; Section 2.2 looks at C++ and why it is the programming language of choice for software-based HFT; Section 2.3 looks at caches and how they are structured; and Section 2.4 looks at the existing software available for identifying and mitigating cache inefficiencies.

Chapter 3 expounds the report's research contribution, aiming to go some way toward filling the gap in the literature with novel investigations looking at a wide range of measurements relating to both data- and instruction-caches and including standard deviation data. Section 3.2 looks at how the structure of data- and instruction-caches affects execution latencies and what this means for programmers writing low-latency software; Section 3.3 looks at contiguous and discontiguous C++ containers, how they are and aren't cache-efficient, and how this affects execution latencies; and Section 3.4 looks at low-level data-layouts, specifically data-member orderings and data-packing, and the effects that these have on cache-efficiency and execution latencies.

Chapter 4 expounds the software contribution, and, in doing so, builds on Chapter 3, presenting a novel piece of software which analyses source-code files and compiled binary files to find cache inefficiencies, alerts the user to these inefficiencies, and suggests mitigations. The chapter first describes the general use of this software in Section 4.1, and then moves on to a detailed exegesis of its development. Section 4.2 details the software's setup phase, Section 4.3 how it performs its cache analysis; Section 4.4 how it analyses user-provided source-code and provides suggestions for mitigating data-cache inefficiencies; Section 4.5 how it analyses user-provided binary files and provides suggestions for mitigating instruction-cache inefficiencies; and Section 4.6 other miscellaneous features.

Chapter 5 evaluates the software contribution in light of the research contribution and Chapter 6 suggests future research directions.

1.4 Summary of achievements

The achievements of this project are manifold. The research contribution provides insight and analysis into the factors underlying cache efficiency with unprecendented detail and replicability. This includes standard deviation measurements and analyses that are crucial to HFT but are almost entirely lacking in the current literature. Consequently, the report is able to provide empirically justified conclusions that challenge existing analyses. Moreover, the software contribution makes previously opaque inefficiencies transparent to programmers, and provides empirically justified suggestions for how to mitigate these inefficiencies in the interests of achieving (a) low execution latencies and (b) a low standard deviation among execution latencies.

Chapter 2

Background

2.1 High Frequency Trading

High Frequency Trading is not particularly easy to define. Aldridge identifies several approaches, contrasting definitions that refer to latency-sensitivity, short position-holding periods, generation of high market-volume with low inventory, a collection of common trading strategies, or behaviour generally beyond the capabilities of humans [1, pp. 13-15]. Jones identifies a number of other possible approaches, including co-location of the trading systems with exchanges, high order-cancellation rates, and ending the day with near-zero positions [30, p. 5]. Given the focus of this report, the most relevant definitions are those that focus on low latencies, and as such I will take High Frequency Trading to be a form of automated (algorithmic) trading whose profitability draws on the capability to react to market data with low latencies.

The impact of HFT on markets is contentious, and so there is a reasonable amount of literature on its economic and legislative effects. Overviews of the economic impact of HFT are given in [31] and [32]; [33] includes an overview of legislation relating to HFT.

Details pertaining to the computational requirements of HFT are less widely discussed for the secrecy and profitability reasons mentioned in Chapter 1. There are, however, presentations at conferences wherein people who work in the industry describe the requirements for building HFT systems and the general setups they run at their respective companies. For example, Goldstein identifies three primary requirements of HFT systems: (1) low latencies, i.e., 5- to 20-microsecond response-times to market data, (2) consistency, i.e., low response-time standard deviations and consistent response-times across different network bandwidths, and (3) reliability, i.e., reliable message generation and resilience to failures of processes and machines [3]. Carl Cook discusses in some detail the requirements for useful simulation and testing of HFT strategies, arguing that in order for tests to be useful it is necessary to consider the impact of optimisations in the context of the entire system from market-data-ingress to order-egress and therefore to build a simulation system that includes a simulated exchange and hardware-timestamping of all messages between the exchange and the trading system [4].

While this report focusses on the optimisation of software-based HFT (which usually operates on the order of microseconds), it is worth keeping in mind that simpler trading strategies (and often network stacks) are sometimes implemented using hardware such as Field Programmable Gate Arrays, which generally operate on the order of nanoseconds [5]. Additionally, given that, as discussed above, the profitability of HFT systems depends on the ability to respond to market data at low latencies, it is straightforward to see why a low standard deviation (low 'jitter') among response times is an important measure and why its absence from the literature ought to be rectified: even if average response latencies are low, individual orders that react to market data slowly run a significant risk of being financially disadvantageous.

2.2 C++

C++ is generally the language of choice for implementing software-based HFT. This is in large part due to the foundational design choices behind the development of C++ which lead to a degree of low-level control not offered by other programming languages. A widely-quoted maxim from Bjarne Stroustrup regarding the development of C++ is that 'what you don't use, you don't pay for' (the 'zero-overhead' principle, cf. [9] and also [8]). In practice, this simple maxim manifests itself in multifarious ways. For example, C++ does not make use of Garbage Collection (GC). Garbage Collectors perform memory management for the programmer, automatically releasing memory when it is no longer needed. This is most commonly achieved by scanning the stack and releasing dynamically allocated memory which can no longer be reached from it. Garbage Collectors, unless they are specifically tuned to do otherwise, may run at any moment, and this causes obvious problems if it happens during the execution of latency-critical code. The absence of GC from C++ means that memory management must be handled by programmers explicitly. This can be challenging, but ensures that latency-critical code execution won't be unexpectedly interrupted. C++ also allows programmers to access and manipulate raw pointers, meaning that the structure and layout of data can be analysed easily and optimised to particular purposes. It is also possible to put assembly language instructions directly inside C++ code. Nevertheless, it is not inconceivable that latency-critical systems might be written in other languages; Aldridge points out that the Nasdaq OMX system is written in Java with GC disabled and direct-memory access capabilities enabled [1, p. 35].

Another general category of features of C++ that render it suitable for ultra-low-latency applications is the wide array of ways in which processing can be transferred from run-time to compile-time. Templates are a versatile tool for this purpose, allowing programmers, for example, to move the run-time overhead of dynamic polymorphism to compile-time at the price of some flexibilility (cf. the 'Curiously Recurring Template Pattern' [34, 35]). Indeed, it has been shown that templates are so versatile as to be Turing-complete [36]. Additionally, keywords like 'constexpr' and 'consteval' can be used to indicate that the compiler should, where possible, evaluate an expression itself rather than leaving it to be evaluated at run-time.

The C++ Standard Template Library makes heavy use of templates to implement a wide variety of common data structures and functions that can hold and operate on data of almost any type [10, 11]. Section 3.3 of this report looks at two kinds of STL container in the context of cache-efficiency: sequence containers and associative containers.

Sequence containers store sequences of items. Beyond this, they are implemented in many different ways. Two well-known kinds of sequence container are 'vectors' and 'lists'; most other classes of sequence container — arrays, deques, queues, forward lists, and the like — are closely related to one or both of these data structures. Vectors hold their elements in contiguous regions of memory, and each element is accessed by calculating its offset from the base of the allocated memory region. It is easy to calculate this offset because the elements are all of the same type and are therefore the same size. Access to an element at a known index is thus achievable in constant time (a constant number of operations). Insertion of an element anywhere other than the end requires shifting elements — in the worst case all the elements in the container — and is therefore achievable in linear time; deletion likewise. If an element is added to the vector and the allocated space is full, a new larger region is allocated and all of its elements (along with the new one) are copied into the new region. This takes linear time but happens increasingly rarely as the vector grows: the allocated space increases exponentially. The converse is true of deleting an element: if possible, the vector will be shrunk by the same factor using the same procedure of re-allocating and copying. Lists, on the other hand, are not stored in contiguous memory; instead, elements may (in principle) be stored anywhere in memory, and each points to the next, forming a chain. Insertion of an element between two known elements simply involves rerouting the chain through the new element, and deletion of an element simply involves making the previous element point to the following element. Each of these involves a constant number of operations and can therefore be achieved in constant time. Access to an element at a known index, however, requires following each of the pointers in the chain of elements the appropriate number of times, and is therefore achievable in linear time.

Associative containers are those which associate the contents of an element itself with its location in the data structure. STL implementations of ordered associative containers (map, set, multimap, and multiset) are required by the C++ standard to guarantee logarithmic time to access, insert, and delete elements of known values and to allow in-order traversal of all the elements in the container. STL implementations of unordered associative containers (unordered map, unordered set, unordered multimap, and unordered multiset) are required by the C++ standard to guarantee constant time to access, insert, and delete elements of known values but are not required to allow in-order traversal. Unordered associative containers (the discussion in Section 3.3 will focus on unordered maps) guarantee constant access time by employing a hash function which selects a 'bucket' in which the element is stored.

It is widely known that STL implementations of these containers have many drawbacks, especially with respect to their cache efficiency. The primary structural culprit behind this inefficiency is their discontiguity in memory. Lists are inherently discontiguous; STL ordered maps are generally implemented as lists oriented as bi-

nary (red-black) trees; and the buckets in STL unordered maps store their elements in lists. Consequently, many implementations of C++ containers that improve on the latency performance of their STL counterparts have been presented. For example, plf::colony (the basis of a current standards proposal called std::hive, cf. [37]) is a data structure which attempts to achieve list-like constant-time erasure and insertion in a contiguous (unordered) data structure. Unlike a vector, which will shift elements backwards after a deletion to fill the gap, a plf::colony will leave the erased slot empty, add it to a list of free slots, and indicate in the slot that it is empty. During traversal, empty fields are skipped; on insertion, elements are inserted into free slots if possible (and to the end if not) [38, 39]. Clearly, if some latency-sensitive code requires frequent erasures and insertions of elements, this can be more efficient than a vector. Unordered maps are a popular target for making latency improvements because the STL unordered map is quite slow. For example, the tsl::hopscotch map [40, 41] uses 'hopscotch hashing' [42] to implement an unordered map in contiguous memory. Simplifying somewhat, hopscotch hashing involves specifying a 'neighbourhood size', and ensuring (by swapping elements around) that elements appear within the neighbourhood of their hash value, limiting the number of contiguous elements that need to be searched to the neighbourhood size. 'Robin Hood hashing' has also been used as a basis for contiguous implementations of unordered maps [43, 44, 45]. In the specific context of HFT, Carl Cook has discussed an 'in-house' contiguous implementation of an unordered associative map that also improves on the STL unordered map [4].

2.3 Caches

A CPU cache is a memory store that is close to, and can be accessed quickly by, the CPU. Memory management systems attempt to predict what data and code are to be used next, and store those data close to the CPU in order that they can be accessed as quickly as possible. Because fast cache memory is expensive, caches are usually organised into 'levels', usually L1, L2, and L3 (and sometimes also L4), each successive level being slower, larger, and further from the CPU. On many modern CPUs, there are distinct L1 data- and L1 instruction-caches. L1 and L2 caches are usually specific to each core, and caches beyond those levels are usually shared between cores. The procedure for accessing memory locations is to check whether data or code from that location are present in the highest level cache, and, failing that (on a 'cache miss'), to check each subsequent cache in order. If the data are not found in any of the caches, the CPU must retrieve it from main memory, or, at worst (which happens when a programme is started and is loaded into memory for the first time), must retrieve it from non-volatile storage such as hard disk drives. Roughly, retrieval from an L1 cache takes about half a nanosecond, retrieval from an L2 cache about seven nanoseconds, and retrieval from main memory about 100 nanoseconds [46]. In more CPU-neutral terms, retrieval from an L1 cache takes about 3 CPU cycles, retrieval from an L2 cache about 14 cycles, and retrieval from main memory about 240 cycles [47]. As such, it is in the interests of low execution latencies that important data are in higher level caches.

Cache memories are divided into cache 'lines' (which are usually 64 bytes long), and most CPU caches are 'N-way set-associative' in structure: each (virtual or physical) memory address is mapped deterministically to a 'set' in the cache (some bits of the address determining the cache set and the lowest order bits of the address determining the offset into the cache line), and within that set the desired data may be in any one of the N ways. (The specific way that holds data from the target address is identified by comparing the target address to the 'tag' on each way in the set.) As such, the number of cache lines is the number of cache sets multiplied by the number of cache ways; the cache can be thought of as a set-by-way matrix of cache lines.

A consequence of the structure of N-way-associative caches is that they have a 'critical stride'. The critical stride of a cache is the distance between memory locations that map to the same set. It is fairly easy to calculate the critical stride of a cache. The size of the entire cache divided by the line size is the number of lines in the cache. (On my processor's L1 data-cache this is 32768 / 64 = 512.) The number of lines per set is the number of ways. (On my processor's L1 data-cache: 8.) The number of sets is the number of lines divided by the number of lines per set. (On my processor's L1 data-cache: 512 / 8 = 64.) The critical stride (the number of bytes between competing locations) is the size of each line multiplied by the number of sets. (On my processor's L1 data-cache: 64 * 64 = 4096.) A less intuitive but also less round-about route to calculating the critical stride is to divide the total size of the cache by its associativity.

Efficient use of caches can be identified in a number of ways. The overarching goal is low execution latencies, but overall latency is a function of more than just cache efficiency. All other things being equal, inefficient use of caches manifests itself in a high percentage of references to those caches resulting in cache misses. Correspondingly, since retrieval of data from lower level caches takes more time and forces instruction execution to wait longer for data retrieval, another indicator of cache inefficiency is a lower number of instructions executed per CPU cycle. However, all other things are often not equal — when comparing the performance of different algorithms, data structures, data layouts, and the like, the (absolute) number of instructions that need to be executed and references to the cache that need to be made may render these percentage measures misleading. In these cases, it may be the case that cache efficiency manifests itself simply in fewer references to the cache. Indeed, fetching data from registers is ideal, taking just one cycle [47], and may be reduced to near-zero latency in certain processors [48]. As such, in the discussion to follow, I will be careful to identify which measures are most appropriate for diagnosing cache inefficiency in each distinct case.

There are a number of widely cited papers on the effects of cache structure on cache-reference behaviours such as how many cache-references result in cache-misses. [12] categorises cache misses into three kinds: (1) 'conflict misses' that arise from different memory locations mapping to the same cache sets (i.e., memory locations that are spaced by a multiple of the critical stride, though they do not use this term), (2) 'capacity misses' that arise because the cache is too small, and (3) 'compulsory misses' that occur when, for example, data is referenced for the first time. [12]

also finds that reducing only the associativity of a cache causes an increase in the percentage of cache-references that result in cache-misses. Relatedly, a number of papers apply these concepts to analysing how conflict (i.e., critical stride) misses can arise from inefficient data-layouts in memory [14, 16], and others to conflict misses arising from inefficient code-layouts in memory [13, 49]. Interestingly, a number of these papers also present software which attempts to solve these problems at various stages, and I will discuss these in the following section.

However, detailed investigation into the effects of cache structure on execution latencies of C++ code that looks beyond surface-level latency characteristics (averaged latencies, averaged cache miss ratios, and so on) is sorely lacking. There are a number of blog posts that describe some very basic experiments pertaining to C++ and cache efficiency ([17, 18, 19, 20, 21]), but, as mentioned above, these rarely venture beyond averaged latencies and also neglect to control for a very large number of factors such as the sizes of the data-types involved, the effects of vector-resizing and copying, and so on. There are also blog posts that compare the performance of large numbers of STL alternatives (e.g., [50]), but these kind of broad rankings are simply descriptive and are unable to provide explanatory depth. Finally, investigation into the effects of data-member ordering and structure packing is few and far between and low on detail; some authors (e.g., [51, 52]) simply assert that data ordering has an effect, and studies into the effects of data-packing find contradictory results (e.g., [18, 20, 21]).

2.4 Existing software

There is software that allows programmers to view the structure of their C++ code once it has been compiled. nm prints the symbols from a compiled object file and has numerous options for printing additional information such as the binary file location, source-code file location, size, and demangled name of each symbol [25, 26]. This allows programmers to see how various components of their code such as functions are laid out in memory when executed. objdump also looks at compiled object files and includes functionality for disassembling these files into human-readable assembly code [23, 24]. gdb is a general purpose debugger which allows programmers to execute programmes step-by-step and is able to describe data-layouts and types-sizes by interpreting debugging symbols [22]. This software is not designed to find inefficiencies, but it is useful for doing so because it enables the examination of memory layouts that are not immediately evident from the source code. Relatedly, there are many vital components of an executable binary file that are implicit in the source code and explicit in compiled code, such as default constructors and virtual function tables.

perf and cachegrind are two more tools that are useful in this regard for different reasons. perf is a profiling tool which runs an executable file, reads values from hardware and software performance counters, and reports these to the user [27]. Importantly, perf gives users the option to run an executable an arbitrary number of times and reports aggregated measures such as standard deviations. cachegrind does not run executable files directly on the CPU (as perf does), but rather simulates their

execution in its own virtual cache environment and evaluates cache-related performance measures such as cache references and cache miss ratios within its simulation [29]. Consequently, unlike perf's results, cachegrind's results are not contingent on the moment-to-moment vagaries of real processors. These tools are clearly useful for comparing the factors underlying the execution latencies of difference pieces of code, but are unable in and of themselves to identify inefficiencies analytically (i.e., non-empirically).

Following on from the discussion on caches above, some software has been developed which aims to improve execution latencies by mitigating data- and instruction-cache inefficiencies automatically. For example, [14] presents a compiler framework that places the stack, heap, global data, and the like in particular virtual memory regions in such a way that data-cache conflicts among them are reduced. Relatedly, [15] presents a related compiler functionality to place code in memory in such a way as to avoid instruction-cache conflicts. While these two papers attempt to mitigate cache inefficiencies by targeting the compilation stage, [16] presents a tool which attempts to mitigate inefficiencies at run-time, placing dynamically allocated data in memory locations in such a way as to avoid data-cache conflicts.

One fundamental problem with these kinds of (automatic) tools for mitigating cache inefficiencies is that they are highly restricted in the solutions they are able to offer. By targeting inefficiencies automatically (and opaquely) at compilation- or run-time, the kinds of solutions that are possible are almost entirely limited to avoiding conflict (critical-stride-based) cache misses by moving data and functions around in memory. However, the range of ways that these kinds of cache-inefficiencies can be mitigated is much wider and more subtle than this. If programmers are alerted to inefficiencies in their code as it would currently compile, there are more options available beside simply moving existing things around. With the high level functionality of the code, the kinds of inputs it receives, and how it integrates with other pieces of code in mind (none of which a compiler or a run-time environment necessarily knows), it is also possible to reorganise data-types, rewrite functions, integrate data-types and functions differently, and so on. Because the existing tools automatically identify and target inefficiencies 'behind the scenes', these kinds of sophisticated optimisations are not possible. For these reasons, the software that I have developed and detail in Chapter 4 aims to make data- and instruction-cache inefficiencies transparent to programmers in order to enable them to mitigate them in deeper and more sophisticated ways.

Chapter 3

Research contribution

3.1 Outline

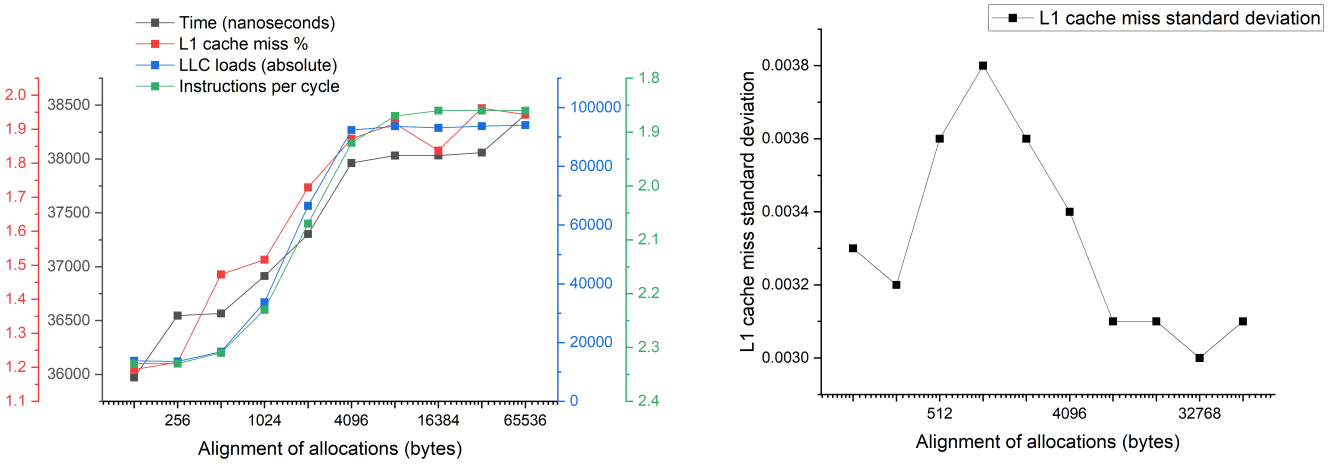
This chapter investigates and illustrates a large number of experiments investigating the ways in which cache considerations contribute to the writing of latency-optimised code. For the purposes of my discussions, latency data has been gathered using Google Benchmark [53], memory allocation data has been gathered using Valgrind [28], and other CPU data has been gathered using perf [27]. (Note that, unfortunately, perf does not distinguish caches beyond L1 and lower-level caches. This is not a major problem because most of the discussion focusses on L1 data- and instruction-caches.) I have been using an Intel(R) Core(TM) i7-8700 CPU, which has 32 kilobyte L1 data- and instruction-caches with linesizes of 64 bytes and 8-way associativities, a 256 kilobyte 4-way associative L2 cache also with a 64 byte linesize, and a 12 megabyte L3 cache. These architecture specifics contribute to the specific numbers at which the relevant effects show up, but the principles are extendable to caches of any related structure. Google Benchmark automatically configures the number of iterations it averages over to produce latency measures in order to balance accuracy with expediency; Valgrind's results do not change iteration to iteration so they are only measured once; the results gathered with perf were averaged over 10,000 iterations per datapoint (with the command-line flag '-r 10000').

3.2 Cache structure effects

This section looks at the effects that the organisation of caches into sets and ways can have on execution latencies.

Data-cache critical-stride effects The first set of tests investigated critical stride effects and involved the following code:

```
#define INT_SIZE sizeof(int)
#define MEM_ALLOCATION 64
#define NUM_ACCESSES 100000
for (int i = 0; i < ptrs; i++)
```



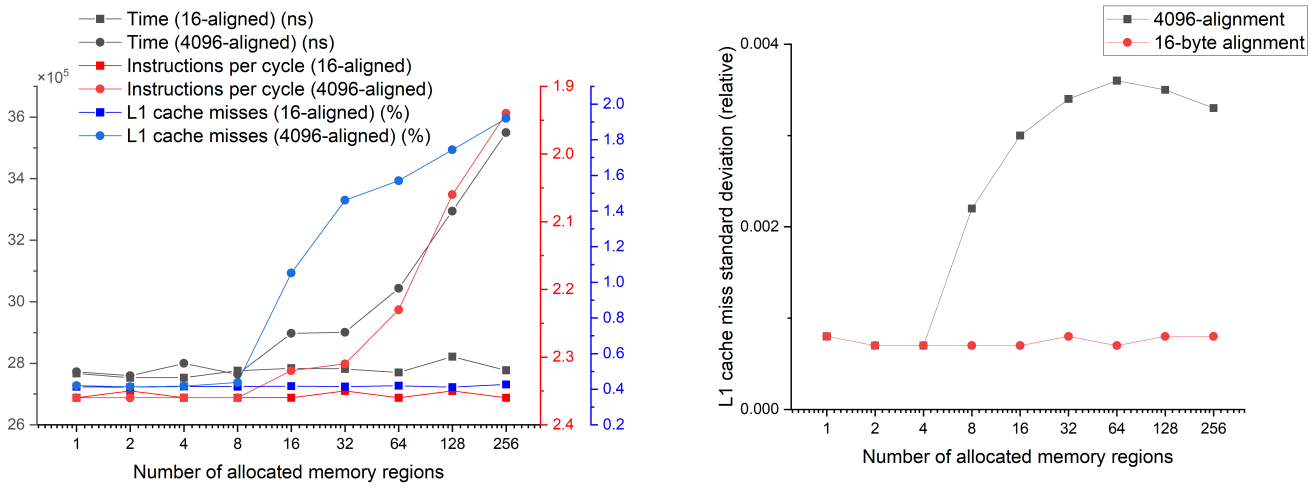
(a) Average values

(b) Cache miss standard deviations

Figure 3.1: Writing and reading data at different alignments

```
{
    posix_memalign((void**) &mem_ptrs[i], alignment, \
                   MEM_ALLOCATION);
}
for (int i = 0; i < ptrs; i++)
{
    for (int j = 0; j < MEM_ALLOCATION / INT_SIZE; j++)
    {
        *(((int*) (mem_ptrs[i])) + j) = (rand() % 100);
    }
}
int dest;
for (int i = 0; i < NUM_ACCESSES; i++)
{
    int ptr_num = rand() % ptrs;
    for (int j = 0; j < MEM_ALLOCATION / INT_SIZE; j++)
    {
        dest = *(((int*) (mem_ptrs[ptr_num]) + j));
    }
}
```

In this test, the independent variable was the value of the variable ‘alignment’, which was cycled through the powers of two from 64 to 65536. The ‘ptrs’ variable was fixed to 256. The posix_memalign() function fills its first parameter with a pointer to allocated memory of the specified size starting on an address that is a multiple of the specified alignment. In each test, 256 regions of memory the size of one cache line were allocated and filled with integers. These regions of memory were then accessed randomly 10,000 times and the integer values read out of them.



(a) Average values

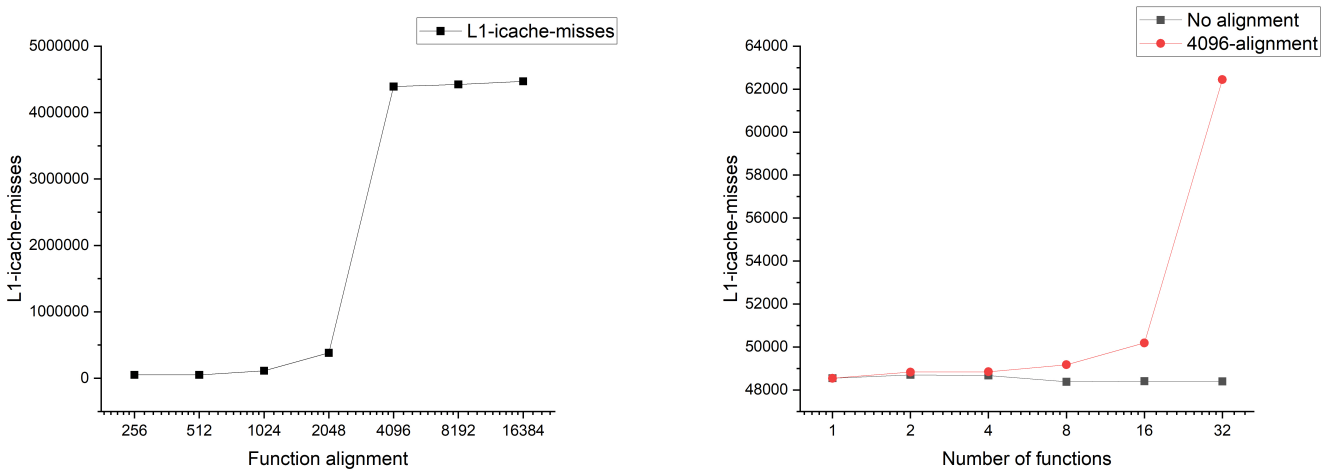
(b) Cache miss standard deviations

Figure 3.2: Writing and reading data in different numbers of aligned/unaligned regions

The results of the tests can be seen in Figure 3.1(a). The pattern of latencies (the dark grey line) that emerges from varying the alignment shows that execution time slows down significantly as the alignment approaches the critical stride and levels out thereafter. Since all things other than the alignment (the number of memory regions, the sizes of the memory regions, and so on) are indeed equal, the number of cache misses as a percentage of references to the cache is a relevant measure of cache efficiency. It is clear from these results that the pattern of cache efficiency by this measure matches the pattern of latencies very closely. This indicates that it is indeed competition for the same cache lines which causes frequent cache misses and references to lower-level (slower) caches, slowing down execution times. Correspondingly, the number of instructions executed per cycle (the green line, note the inverted axis) shows the same pattern, as does the absolute number of references made to the lower level caches.

A further interesting result from this first set of tests (and one which, as far as I can ascertain, has not been presented anywhere else in the literature) can be seen in Figure 3.1(b). This figure demonstrates that, at alignments leading up to the critical stride, the standard deviation in the number of cache misses is comparatively high. This is because at such alignments there will be an increasingly high probability (approaching 100 percent at the 4096 threshold) that two memory regions will compete for a cache set. As mentioned above, in the context of HFT this standard deviation measure is of critical importance.

Data-cache associativity effects The second set of tests investigating critical stride effects involved the same code as the first set of tests, but this time the number of pointers (the variable 'ptrs', and, correspondingly, the number of memory regions allocated) was cycled through the powers of two from one to 256 and the 'alignment' variable was restricted to two values, 16 (non-critical-stride) and 4096 (critical-



(a) Instruction cache misses (varying alignments) **(b)** Instruction cache misses (varying number of functions)

Figure 3.3: Instructions at varying alignments and in varying numbers of functions

stride).

The results of this test can be seen in Figure 3.2(a). The pattern of latencies (the dark grey lines) shows that above (but not including) eight memory regions, data aligned by 4096 bytes is written to and accessed slower than data aligned by 16 bytes. However, below (and including) eight memory regions, there is no difference between 16- and 4096-byte-aligned data. This result arises because of the associativity of the cache. The 8-way associativity of the processor I ran these tests on means that eight 64-byte regions can be maintained in each cache set at any time. If there are any more than eight regions, some data in the cache will have to be evicted to make space for the new data, and the next time that data needs to be accessed the reference will miss the cache, causing a slow-down in execution speed. Again, cache misses as a percentage of cache references is a close predictor of this latency result, as is the number of instructions executed per cycle (note again the inverted axis).

It is also important to consider standard deviations in this case. The standard deviation in cache misses can be seen in Figure 3.2(b). While the averaged results in Figure 3.2(a) appear to show that there is no problem with eight memory regions aligned on the critical stride, this result demonstrates that there is significant jitter on this threshold.

Extending the results to the instruction-cache Although the discussion so far has focussed on the data-cache, the principles can be extended to the instruction-cache. On the processor I have been using, it happens to be the case that the L1 instruction-cache has exactly the same structure as the L1 data-cache. Correspondingly, similar results can be observed when code is aligned in particular ways. For example, I used the gcc flag -falign-functions=... to force functions of the following (very simple) kind onto particular alignments:

```
void function_1()
```

```
{
    int i = 0;
    for (int j = 0; j < 10; j++)
    {
        i = j + 1;
    }
}

void function_2()
{
    int i = 0;
    for (int j = 0; j < 10; j++)
    {
        i = j + 1;
    }
}
...(etc.)...
```

The result of this compilation option on a programme with 32 such functions can be observed with the nm command (specifically, 'nm -C -v path/to/programme'). Here is the output of nm called on a programme compiled without function alignment:

```
0000000000001000 t _init
0000000000001080 T _start
00000000000010b0 t deregister_tm_clones
00000000000010e0 t register_tm_clones
0000000000001120 t __do_global_dtors_aux
0000000000001160 t frame_dummy
0000000000001169 T function_1()
0000000000001197 T function_2()
00000000000011c5 T function_3()
00000000000011f3 T function_4()
0000000000001221 T function_5()
...(etc.)...
```

And here is the output of nm called on a programme compiled with functions aligned to 4096-bytes:

```
0000000000001000 t _init
0000000000002000 T _start
0000000000002030 t deregister_tm_clones
0000000000002060 t register_tm_clones
00000000000020a0 t __do_global_dtors_aux
00000000000020e0 t frame_dummy
0000000000003000 T function_1()
0000000000004000 T function_2()
0000000000005000 T function_3()
0000000000006000 T function_4()
```

```
0000000000007000 T function_5()
...(etc.)...
```

nm can therefore be used by programmers to identify functions which may compete for space in the cache. The effect of the gcc flag on the assembly can be seen from the padding after the end of function_1() (using objdump, specifically 'objdump -d -S -C -Mintel –no-show-raw-insn path/to/programme', requesting Intel syntax simply because I am more familiar with it than I am with AT&T syntax!):

```
302c:   pop    rbp
302d:   ret
302e:   jmp    4000 <function_2()>
3033:   data16 nop WORD PTR cs:[rax+rax*1+0x0]
303e:   data16 nop WORD PTR cs:[rax+rax*1+0x0]
3049:   data16 nop WORD PTR cs:[rax+rax*1+0x0]
3054:   data16 nop WORD PTR cs:[rax+rax*1+0x0]
...(etc.)...
```

These functions were then called in sequence in a loop as follows:

```
for (int i = 0; i < NUM_CALLS / NUM_FUNCTIONS; i++)
{
    function_1();
    function_2();
    ...(etc.)...
}
```

When keeping the number of functions constant and varying the alignment, the pattern in Figure 3.3(a) was observed. When the number of functions was varied and 16-byte and 4096-byte alignments were compared, the pattern in Figure 3.3(b) was observed.

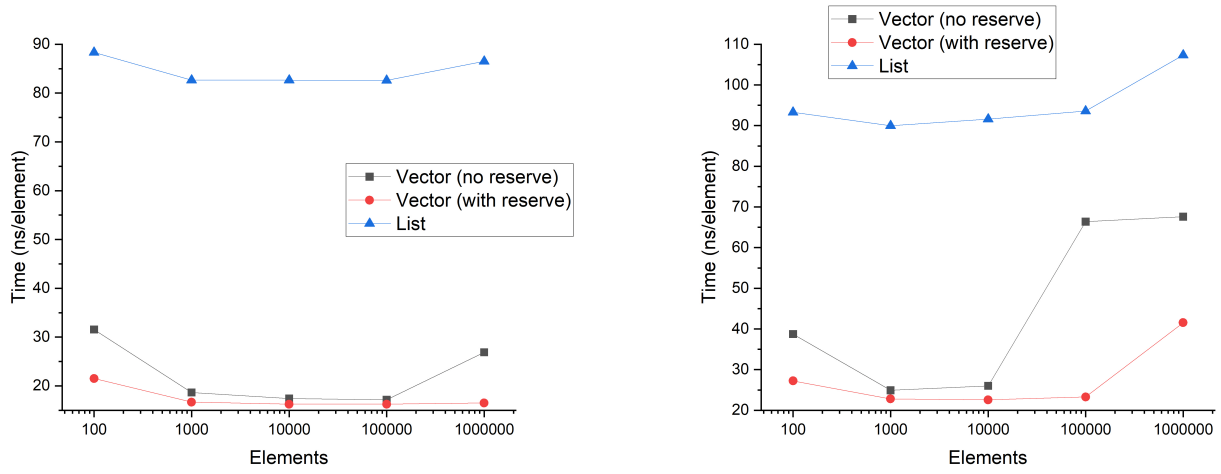
Interim conclusions From the first set of alignment tests (relating to the data-cache critical stride), it can be concluded that to increase execution speed it is important to make sure that data does not compete for cache space. In order to do this, data whose access patterns coincide must not be spaced by a multiple of the critical stride, or, ideally (to reduce jitter), any high factor of the critical stride. The results of the second set of alignment tests (relating to the data-cache associativity) add nuance to the results of the first set of tests. While overlapping data use on critical-stride alignment can indeed have a significant deleterious effect on latency, this effect may be mitigated if the number of such regions falls below the number of ways in the cache. However, falling within this boundary may also not be entirely satisfactory on its own, because jitter increases as the number of competing regions approaches the number of cache ways. From the third set of alignment tests (relating to the instruction-cache), it can be seen that the structure of the instruction-cache forces the same considerations on programmers that the data-cache does.

In short, if enough data- or instruction-sequences compete for cache lines as a result of mapping to the same cache sets, there can be a significant reduction in

latency compared to data-/instruction-layouts that do not result in such cache conflicts. Indeed, every dimension of the structure of a cache — its overall size, the size of each cache line, the associativity of the cache, and so on — has an effect on how programmers should go about writing maximally efficient code, and programmers should therefore be very mindful of the structure of the caches on the processor on which their code is intended to run. Of course, as will become clearer in later sections, there is some conflict between the advantages of avoiding high-multiple alignment and the advantages of avoiding low-multiple (un)alignment. As such there is a sort of alignment sweet-spot that programmers should aim to hit in order to use the cache as efficiently as possible.

3.3 C++ containers

This section looks at the relative latencies of operations which, in theoretical terms, have the same algorithmic complexity, but which perform differently on real processors as a result of cache-structure and cache-reference behaviour.

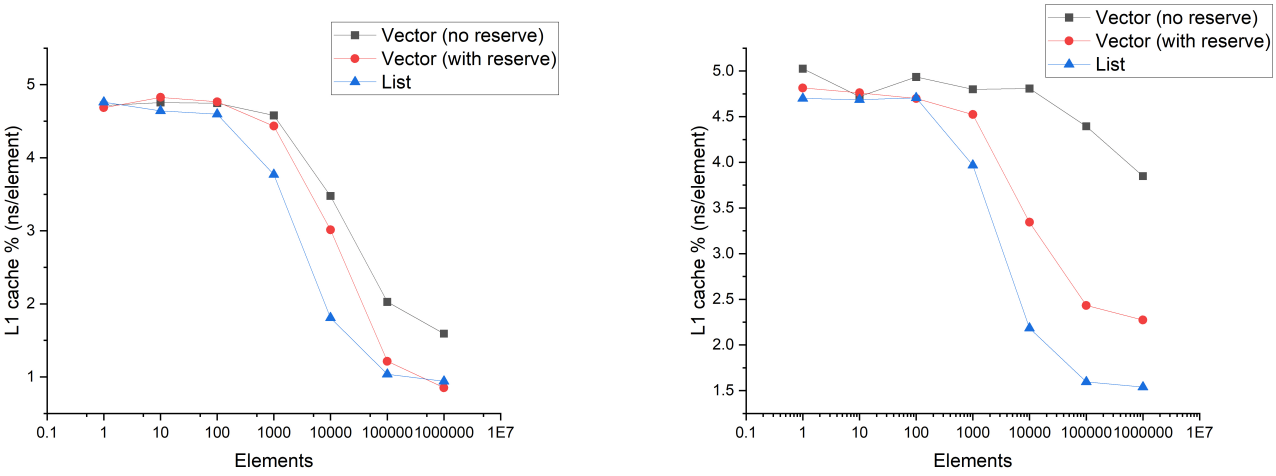


(a) 16-byte elements

(b) 64-byte elements

Figure 3.4: Insertion of items to vector/list vs time

Sequence containers The sequence containers involved in these tests were vectors and lists. The first set of tests involved appending elements to the end of each container and then iterating through those elements in order. Adding an item to the back of a sequence container takes constant time for a list and (amortised, cf. resizing) constant time on a vector. Likewise, traversal of all elements of the container takes linear time for each. However, while theoretically comparable, the structure of these containers in memory (cf. Section 2.2) has consequences for memory efficiency and hence latency. The code for this test was as follows:



(a) 16-byte elements

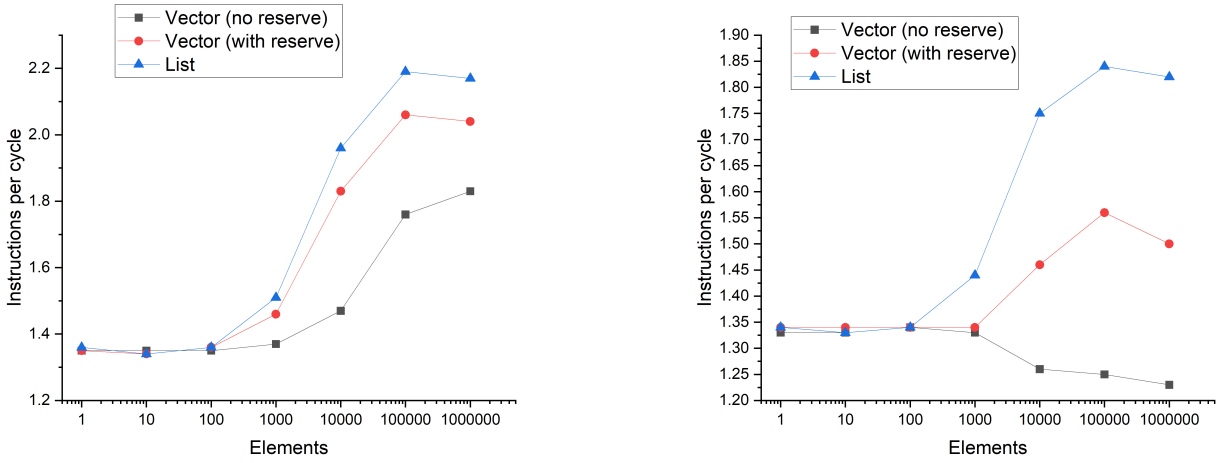
(b) 64-byte elements

Figure 3.5: Insertion and retrieval of items to vector/list vs cache miss %

```
#define BYTES 16 [or 64]
struct D
{
    int i[BYTES / 4];
};
(...)
for (int i = 0; i < num_elements; i++)
{
    D d;
    d.i[0] = rand_num;
    d.i[1] = rand_num;
    d.i[(BYTES / 4) - 2] = rand_num;
    d.i[(BYTES / 4) - 1] = rand_num;
    container.push_back(d);
}
D dest;
for (auto i : container)
{
    dest = i;
}
```

Algorithmically speaking, these operations have the same time complexity for both types of container; however, the latency results are quite different. Figure 3.4 shows the time taken (per element) to add a varying number of 16-/64-byte elements to the end of each kind of container (also comparing the performance of a vector which reserves adequate space in advance with a vector which does not).

Clearly, although identical in theoretical time complexity, the two containers perform quite differently, regardless of whether the element takes up a full cache line



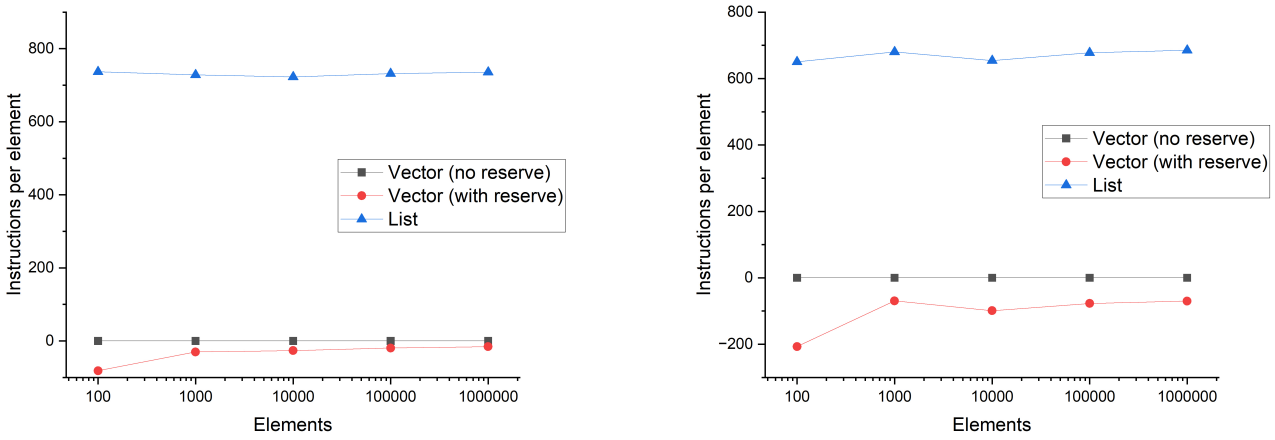
(a) 16-byte elements

(b) 64-byte elements

Figure 3.6: Insertion and retrieval of items to vector/list vs instructions-per-cycle

or not. However, when one starts to look for explanations in the expected places, answers are not forthcoming. Counterintuitively, Figures 3.5(a) and 3.5(b) show that lists actually show the lowest percentages of cache misses. Figures 3.6(a) and 3.6(b) likewise show that lists show the highest numbers of instructions executed per cycle.

At first thought these results might seem unexpected, but given that the process of allocating and constructing a list node is significantly different to allocating and constructing a new element in a vector, it makes more sense to look at the absolute numbers of instructions and cache references. Indeed, Figures 3.7 and 3.8 clearly show that the source of the slower execution speeds when appending to and traversing a list is the sheer number of instructions that must be executed and references to cache memory that must be made. (Note that these graphs show instructions/cache loads (a) per element and (b) relative to vector-without-reserve. Without these two normalisations, the numbers were too many orders of magnitude apart to distinguish the lines!) On reflection, this makes sense: allocating and constructing an element in a list involves allocating new memory for the new element, updating the next-element pointer in the preceding element, updating the pointer to the tail of the list, setting the pointers in the newly allocated node, and filling the new node with the relevant data itself. Traversing a list involves checking the next-element pointer (which is hopefully in the cache) to determine the location of the following element. On the other hand, no such overhead is required with a vector. To append an item to or to traverse a vector requires only calculating offsets from the base of the vector, which can be performed using simple in-register calculations and requires very few (if any) references to the cache. A detailed look at the assembly code involved in the push_back() function for a vector and a list demonstrates how significant the difference in processing required to append a new element is. As a simple indication, the list's push_back(D const&) function includes a call to a func-



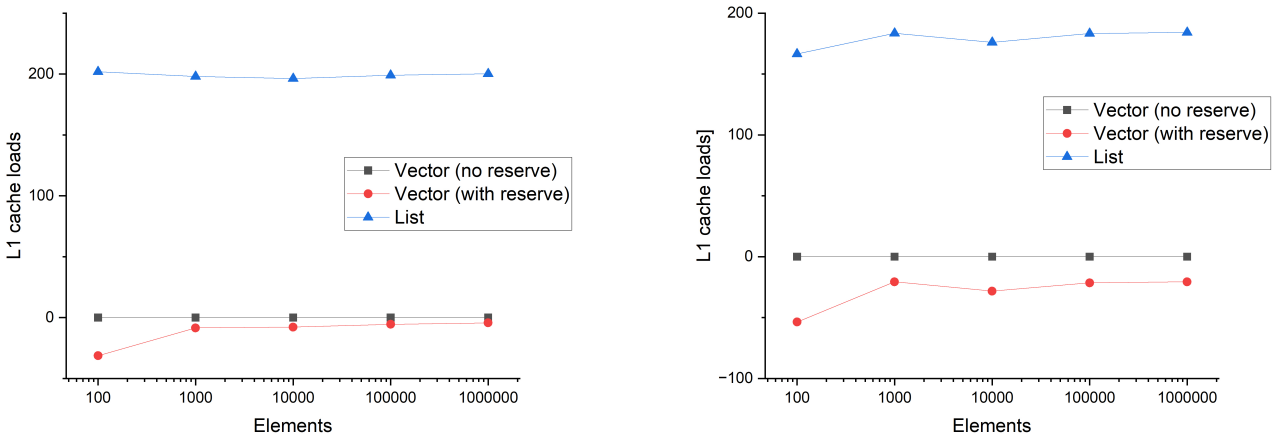
(a) 16-byte elements

(b) 64-byte elements

Figure 3.7: Insertion and retrieval of items to vector/list vs raw instruction numbers (per element, relative to vector-without-reserve)

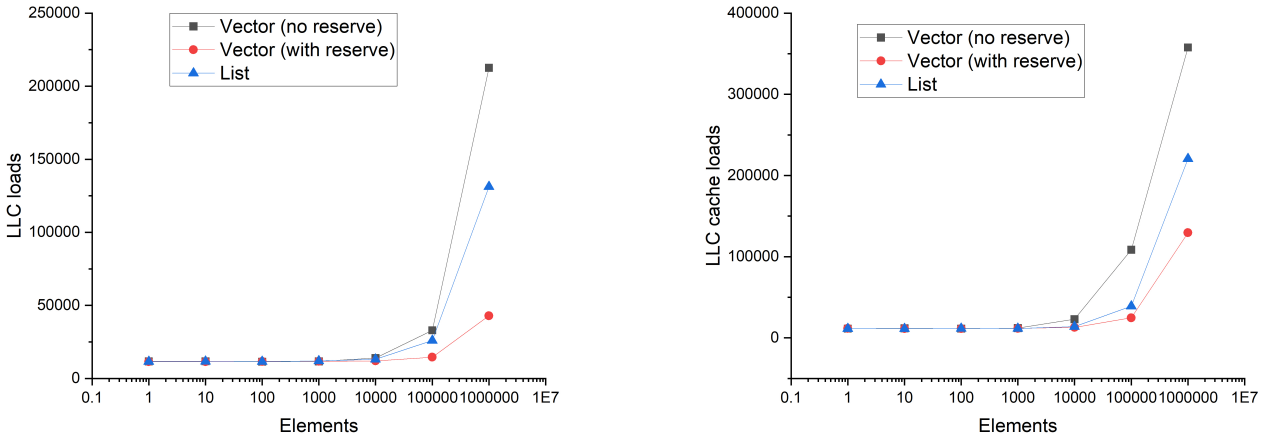
tion _M_insert, which itself involves calls to _M_create_node, which involves calls to functions called _M_get_node, _M_get_Node_allocator, __allocated_ptr, and only then is the construct function which actually builds the new node called. The vector requires none of these myriad nested function calls; its push_back function (assuming no resizing) calls the construct function directly. As alluded to above, the most cache efficient structure is one which requires no references to the cache at all!

However, there is one noticeable pattern in the latency data that has not yet been explained by the instruction and cache load data: the upward jump in time per element in all cases when 1,000,000 elements are involved, and, interestingly, in the specific case in Figure 3.4 of 100,000 64-byte elements in the vector that does not reserve space in advance. When looking carefully at the data pertaining to absolute numbers of references to lower level caches (L2 and L3 caches), it is clear why this pattern arises, and how, once again, latency patterns are a function of cache-efficiency. In Figure 3.9 it can be observed that the absolute number of lower level cache references spikes earlier in the specific case of a vector which does not reserve space in advance and which contains elements that are 64-bytes in size. Since lower level cache references are particularly expensive in terms of latency, the pattern observed in Figure 3.4 is of no surprise.



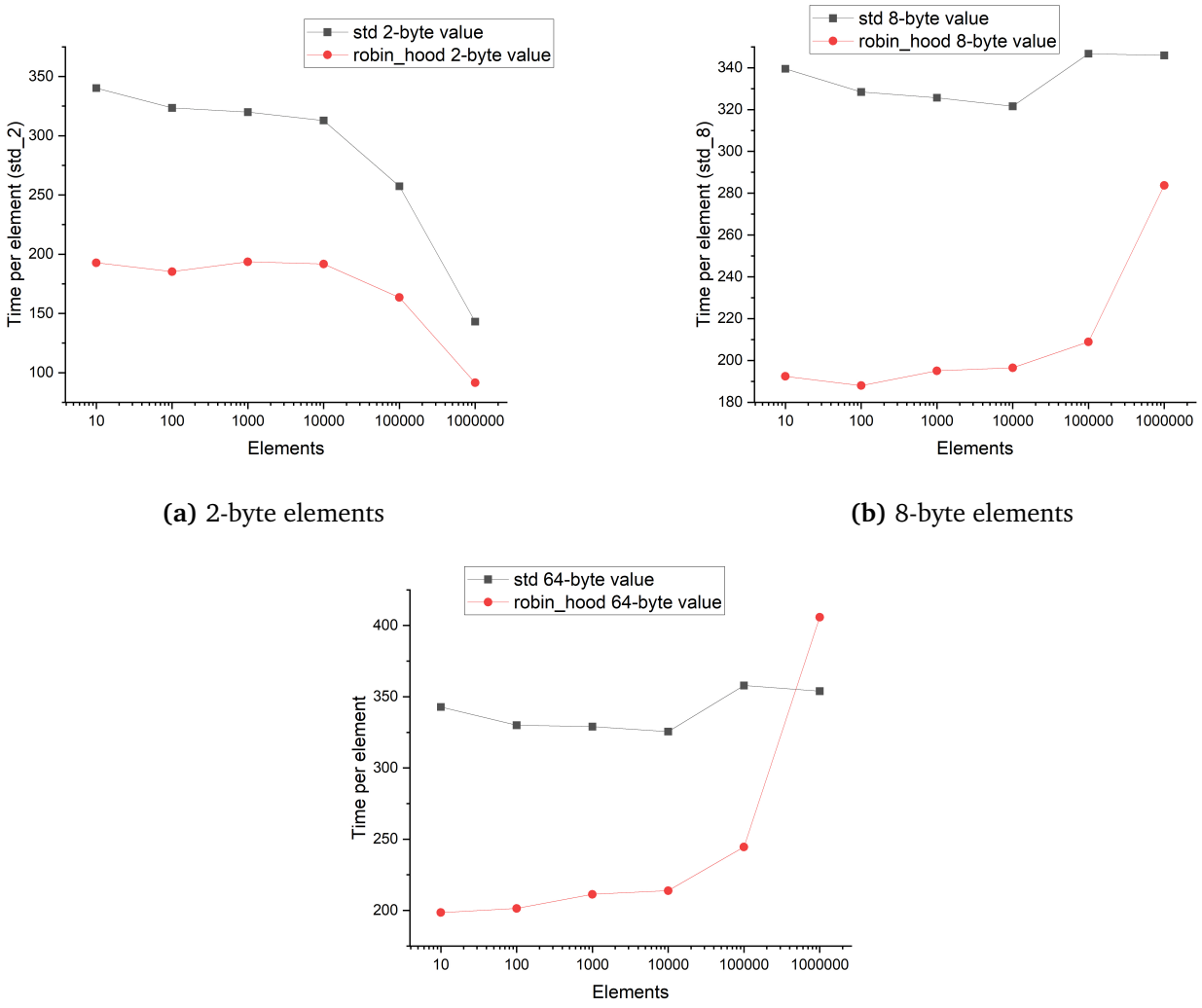
(a) 16-byte elements **(b)** 64-byte elements

Figure 3.8: Insertion and retrieval of items to vector/list vs raw cache load numbers (per element, relative to vector-without-reserve)



(a) 16-byte elements **(b)** 64-byte elements

Figure 3.9: Insertion and retrieval of items to vector/list vs lower-level cache loads



(a) 2-byte elements

(b) 8-byte elements

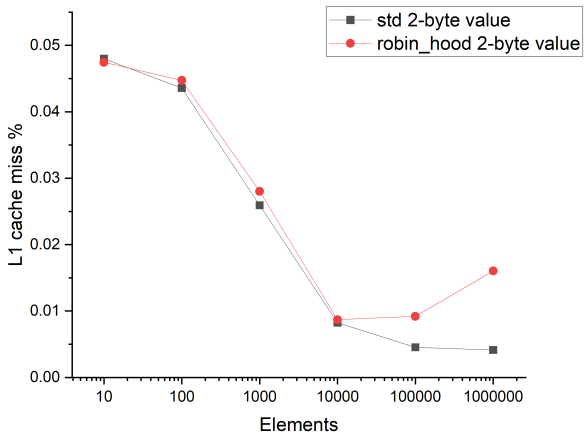
(c) 64-byte elements

Figure 3.10: std vs robin_hood unordered map insertion and access vs time per element

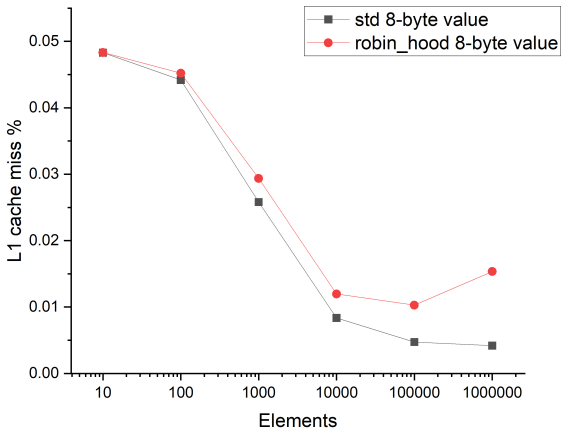
Associative containers In the tests that follow, a contiguous unordered map implementation based on 'Robin Hood hashing' has been used. Note that this is not meant to indicate that this kind of implementation is any better or worse than any other contiguous-memory implementation, just that it is easy to use and that Robin Hood hashing is long-established and well-understood.

In order to test the consequences of using contiguous storage to implement unordered maps, the following code was used:

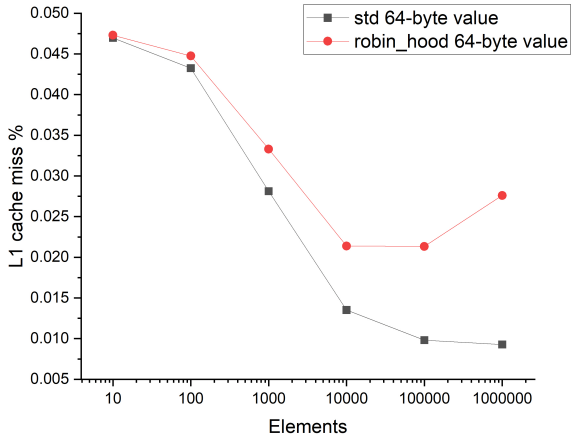
```
for (int i = 0; i < num_elements; i++)
{
    m[i].i = rand_num;
}
int dest;
```



(a) 2-byte elements



(b) 8-byte elements



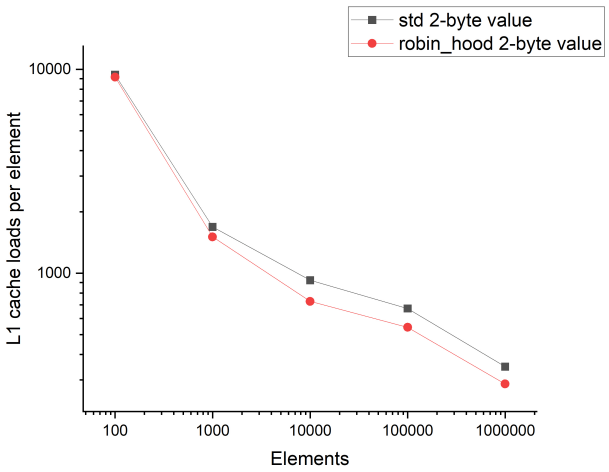
(c) 64-byte elements

Figure 3.11: std vs robin_hood unordered map insertion and access vs cache miss %

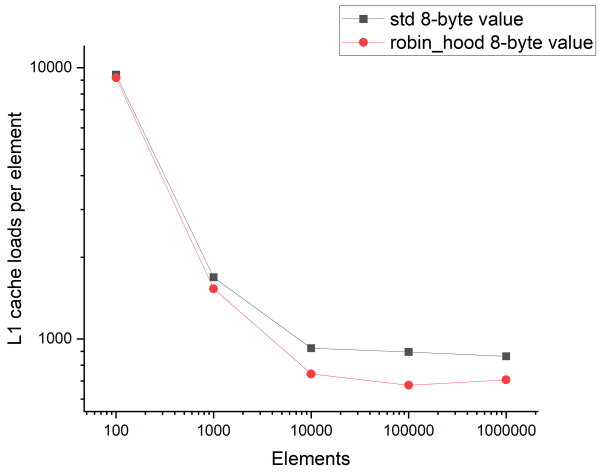
```
for (int i = 0; i < num_elements; i++)
{
    dest = m[i].i;
}
```

All data was collected three times with differently sized elements in the container each time, using the following (small-/medium-/large-value) declarations with a cache-line sized struct for the largest type:

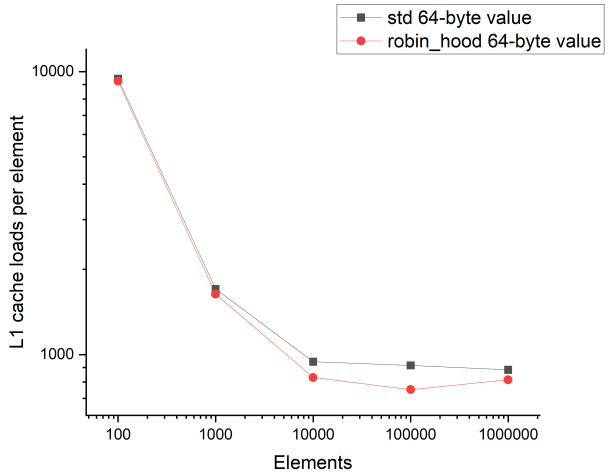
```
struct bytes_64
{
    int i:512;
};
std::unordered_map<uint16_t, uint16_t> m;
std::unordered_map<uint64_t, uint64_t> m;
```



(a) 2-byte elements



(b) 8-byte elements



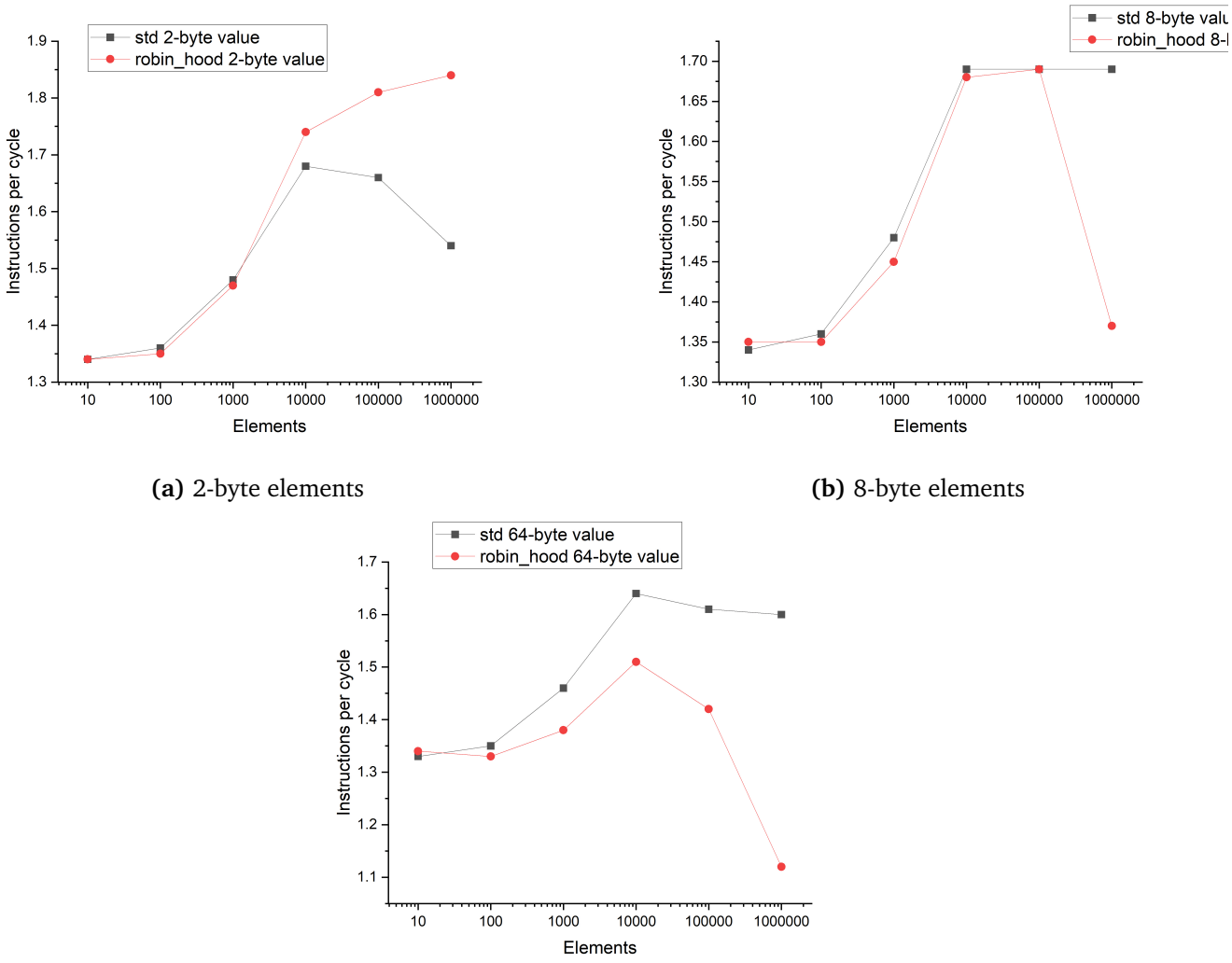
(c) 64-byte elements

Figure 3.12: std vs robin_hood unordered map insertion and access vs cache loads

```
std::unordered_map<uint64_t, bytes_64> m;
robin_hood::unordered_flat_map<uint16_t, uint16_t> m;
robin_hood::unordered_flat_map<uint64_t, uint64_t> m;
robin_hood::unordered_flat_map<uint64_t, bytes_64> m;
```

The first set of tests involved varying the value of 'num_elements', i.e., the number of elements which were inserted into the unordered map and then read out. The latency results can be seen in Figure 3.10.

The clearest outcome is that, in almost all cases, the unordered map implementation using contiguous memory is significantly faster than the STL implementation using discontiguous memory. In fact, as might be expected, a great number of the results mirror those found in the sequence-container tests, and therefore develop and strengthen the conclusions that can be drawn from them. For example, the data relating directly to cache performance are strongly reminiscent of the vector-versus-



(a) 2-byte elements

(b) 8-byte elements

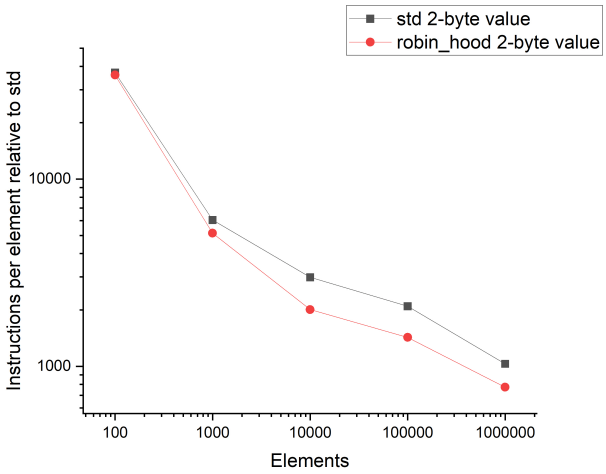
(c) 64-byte elements

Figure 3.13: std vs robin_hood unordered map insertion and access vs instructions per cycle

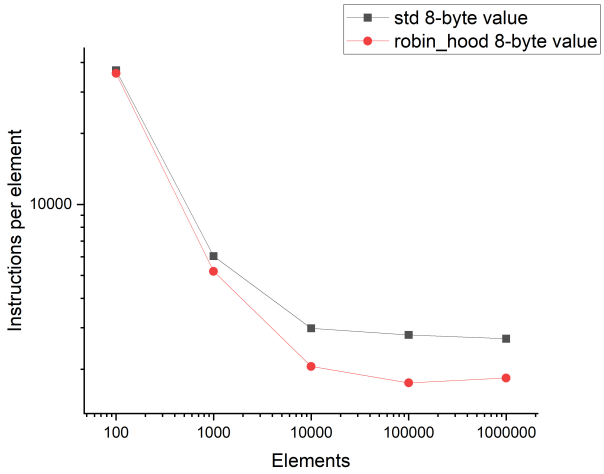
list data: the robin_hood (contiguous) unordered maps unexpectedly show a higher cache miss percentage, as seen in Figure 3.11, but consistently lower numbers of cache references overall, as seen in Figure 3.12.

The instructions-per-cycle data show less clear-cut results. From Figure 3.13 it can be seen that there is no consistent relationship between the instructions-per-cycle when processing the STL map compared to processing the robin_hood map. Regardless, the pattern of latencies is again best predicted by the absolute numbers of instructions executed per element, which can be seen in Figure 3.14. The instructions-per-element data show remarkable correspondence to the cache-loads-per-element data, and both closely predict the latency data.

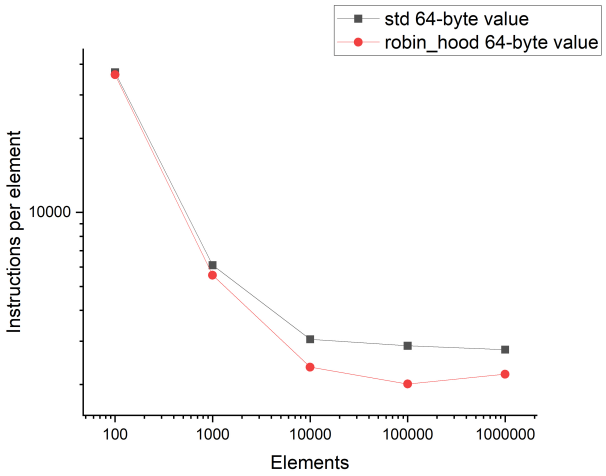
Again, similarly to the vector-versus-list data, there is an outstanding pattern in the latency data to explain: the large spike in latency for containers with 1,000,000 elements. When the elements are 64 bytes in size, this spike actually makes the



(a) 2-byte elements



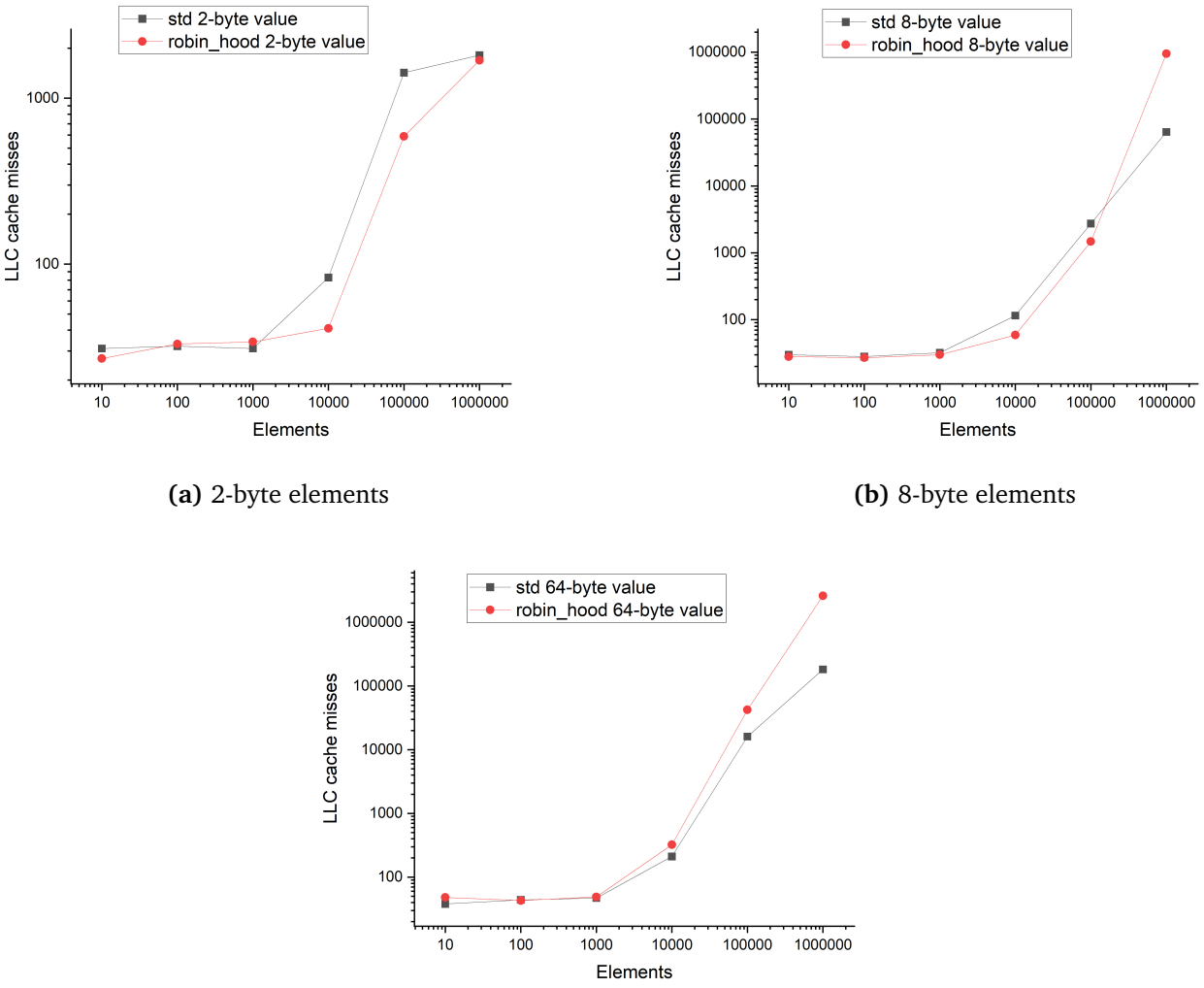
(b) 8-byte elements



(c) 64-byte elements

Figure 3.14: std vs robin_hood unordered map insertion and access vs instructions per element

robin_hood map slower than the STL map. Unlike the sequence container data, however, this result does not appear to emerge from the number of references to the lower level caches, but rather from an even more extreme source of execution slow-down: lower level cache misses. It can be seen from Figure 3.15 that for lower numbers of elements, the number of lower level cache misses for the standard map and the robin_hood map are consistently in the same order of magnitude (emphasised by the logarithmic Y-axis). However, for larger numbers of elements — and specifically where the latency of the robin_hood map jumps sharply upward — the number of lower level cache misses shown by the robin_hood map is an order of magnitude larger than that shown by the STL map. This is strong evidence that the latency spike is a result of the fact that the data structure is too large to use the lower level caches efficiently. Clearly, if many elements are to be stored in a particular data structure, it is of great importance to the programmer of low-latency software to



(a) 2-byte elements

(b) 8-byte elements

(c) 64-byte elements

Figure 3.15: std vs robin_hood unordered map insertion and access vs lower-level cache misses

consider the space efficiency of their data structures and whether their data will fit in their processor caches.

Interim conclusions The data presented that pertains to C++ containers lead to both methodological and practical conclusions. The methodological conclusion is that, as mentioned above, the most obvious measures of cache inefficiency — cache miss percentage and instructions executed per cycle — are inappropriate in more complex situations. When there is a significant discrepancy in the absolute numbers of instructions that need to be executed or the numbers of (especially lower level) cache references that need to be made, these will be the determining factors.

The practical conclusion that can be drawn is that abstract measures of algorithmic complexity can only get you so far; programmers have to consider the per-

formance in terms of actual computer architectures. The goal of the programmer who programmes for low latencies is not to reduce algorithmic complexity in the abstract sense but to reduce the number of instructions that must be executed and to reduce the number of references that must be made to the cache. Of course, for obvious reasons, an algorithm with better theoretical time complexity will very often also involve fewer instructions and fewer cache references. However, this is not necessarily the case, and it has been observed that algorithms that are notoriously bad in terms of theoretical time complexity (such as bubble sort) are actually faster in many common use cases than algorithms whose time complexity is theoretically better [6]. The important conclusion is therefore that the primary concern is efficient organisation and use of data and instructions in the cache, and that optimising algorithmic time complexity is often, but not always, in service of this goal.

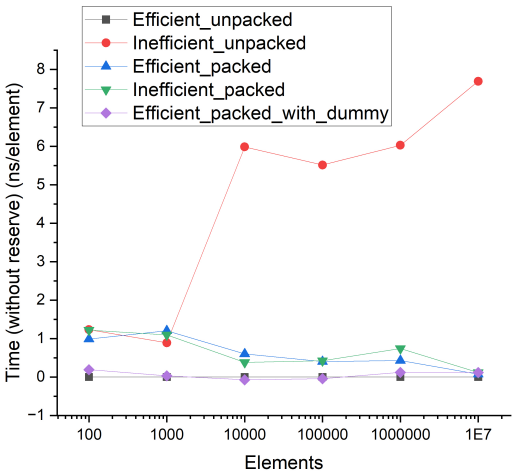
3.4 Data layouts

Data ordering and packing The fourth and final set of tests relating to cache efficiency pertains to the ordering, padding, and alignment of data members in user-defined types, and the complex interactions of these factors. These tests have produced the most interesting and complex results, none of which have been explicated in as much detail as what follows, and many of which have apparently not before been documented at all and yet have significant ramifications for the writing of cache-efficient code.

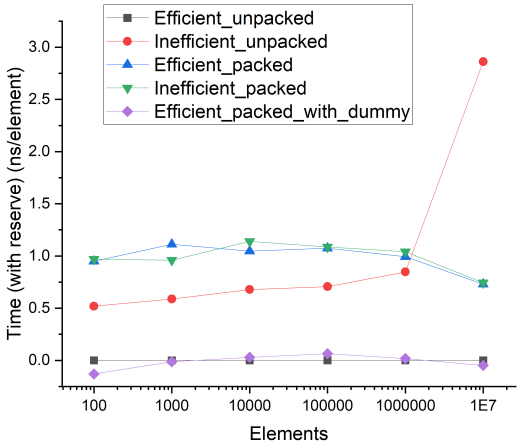
It is often simply asserted that data ordering has an effect on latency [51, 52], but there is disagreement as to whether (small-multiple) data (un)alignment does [18] or doesn't [20, 21] have an effect on latency. The discussion that follows provides unprecedented detail and attempts to clarify some of the answers to these questions.

Data ordering is extremely simple. The following shows an example of an efficiently ordered and an inefficiently ordered struct with exactly the same data members:

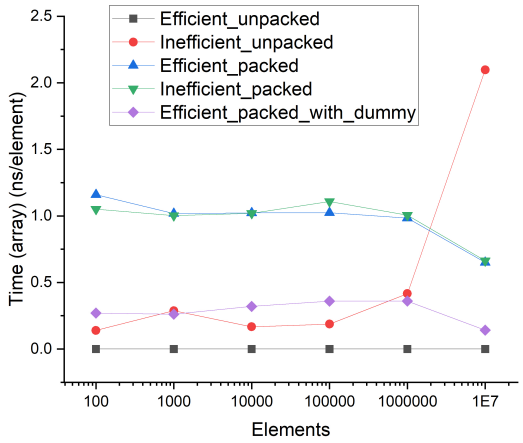
```
struct Efficient_unpacked
{
    char c;
    short s;
    int i;
    double d;
};
struct Inefficient_unpacked
{
    char c;
    double d;
    short s;
    int i;
};
```



(a) Vector without reserve



(b) Vector with reserve



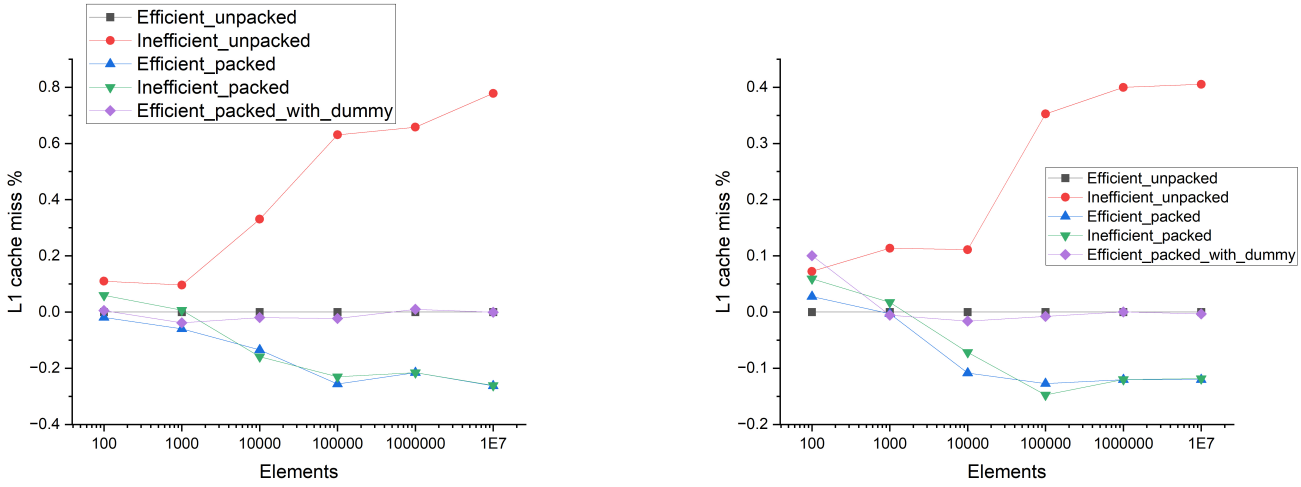
(c) Array

Figure 3.16: Latency per element for appending to and reading from a vector/array

Each data member is aligned on multiples of its size, and so in the efficient ordering there is one byte of padding between the char and the short. In the inefficient ordering, there are seven bytes of padding between the char and the double. As such, the efficient struct has total size 16 bytes and the inefficient struct 24 bytes.

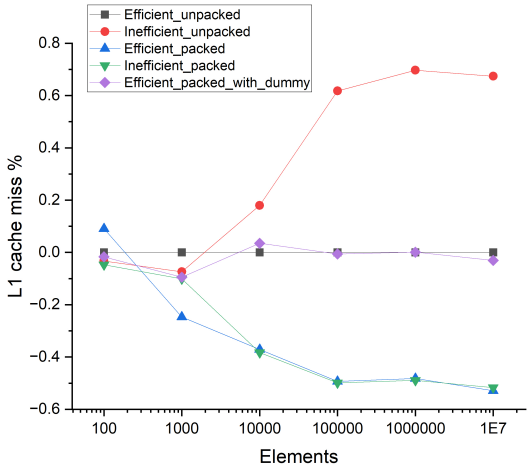
However, it is possible to remove the padding entirely and 'pack' the data as tightly as possible, forgoing alignment; gcc allows this with __attribute__((packed)). (Note that on some ARM processors, the unaligned accesses that result from packing can crash the programme or lead to undefined behaviour [54]. As such, the question of whether programmers should pack data structures is only relevant to those using processors that allow unaligned accesses. The other questions are relevant to all C++ programmers.) Each of the following structs is 15 bytes in size:

```
struct Efficient_packed
{
```



(a) Vector without reserve

(b) Vector with reserve

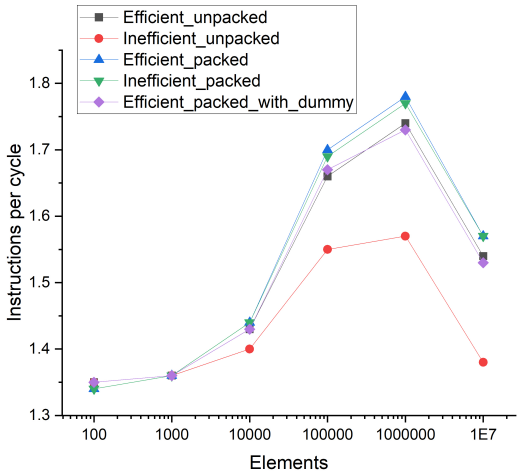


(c) Array

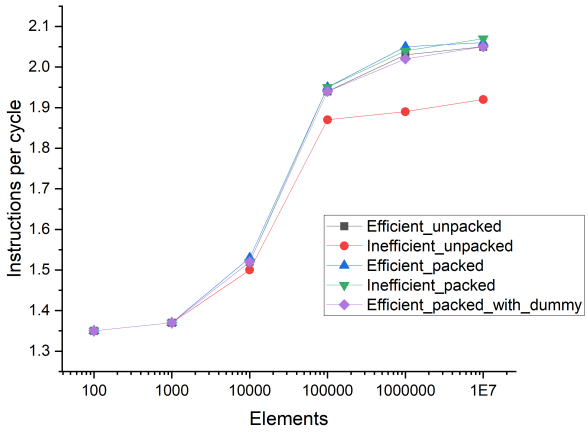
Figure 3.17: Cache miss % for appending to and reading from a vector/array

```
    char c;
    short s;
    int i;
    double d;
} __attribute__((packed));
struct Inefficient_packed
{
    char c;
    double d;
    short s;
    int i;
} __attribute__((packed));
```

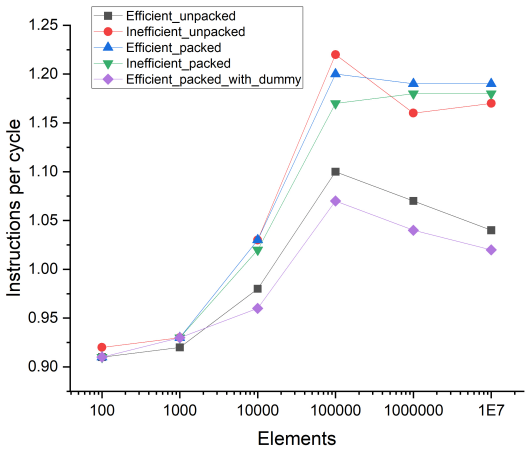
These possibilities raise many questions. First, one can ask whether the simple or-



(a) Vector without reserve



(b) Vector with reserve



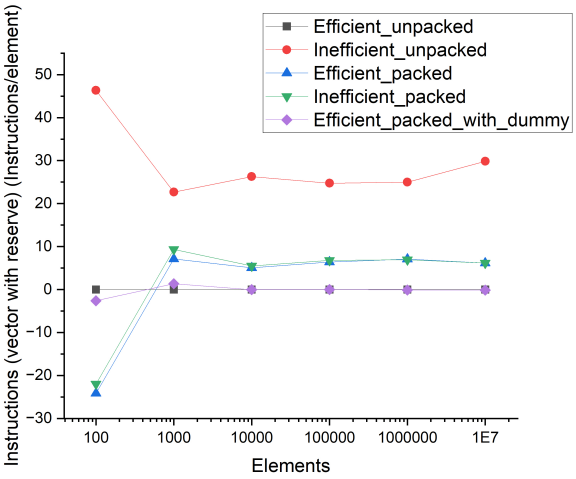
(c) Array

Figure 3.18: Instructions per cycle for appending to and reading from a vector/array

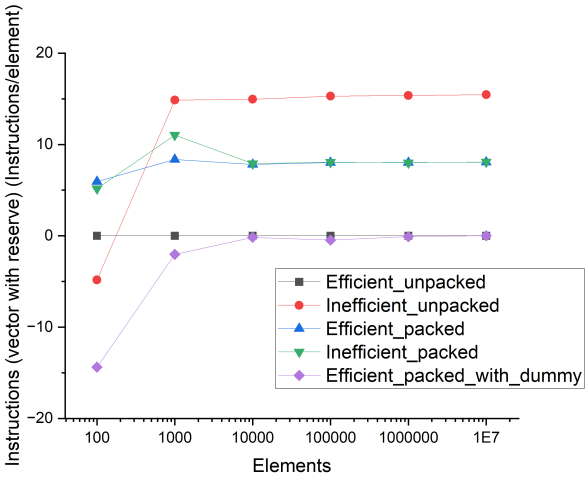
dering of data members in a struct has an effect on the efficiency of the code that uses it. Following up on this question, it is also pertinent to ask whether the size advantages of packing efficient and inefficient structs corresponds to an advantage in terms of latency when accessing and/or copying the data. On the other hand, it is also worth asking whether the misalignment of data that packing entails creates any latency problems. If both are the case, it would be useful to know which factor is dominant.

To make some progress towards answering these questions, a series of tests were run involving these different kinds of struct. Each of these tests involved appending some number of elements to the end of a vector or array and then reading each one of those elements out into a destination object of the same type, as follows (for a vector):

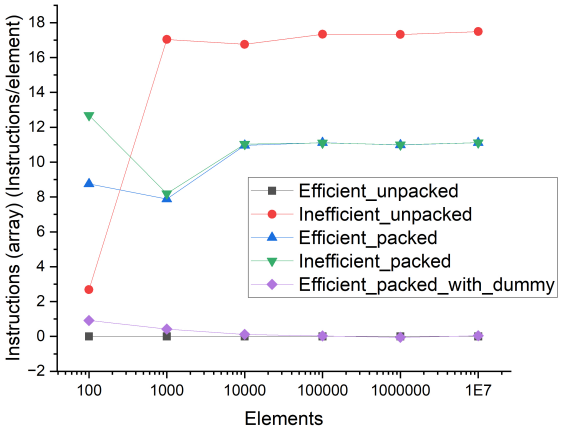
```
for (int i = 0; i < num_elements; i++)
```



(a) Vector without reserve



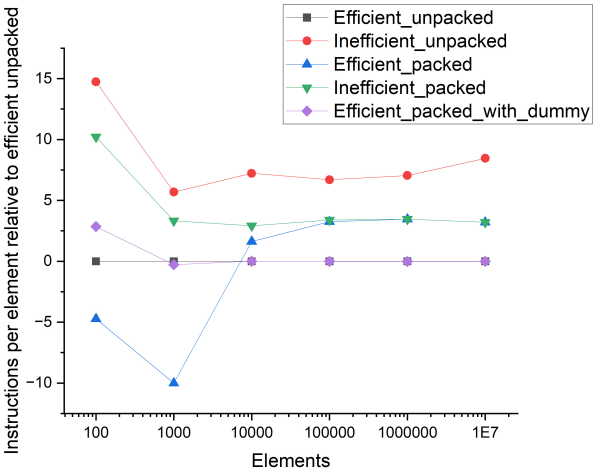
(b) Vector with reserve



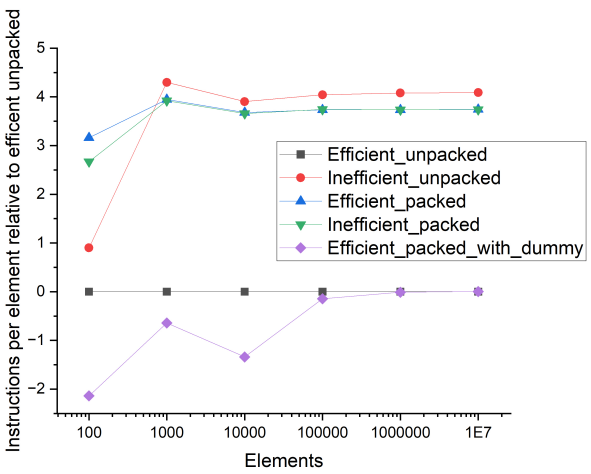
(c) Array

Figure 3.19: Instructions per element (relative to efficient unpacked) for appending to and reading from a vector/array

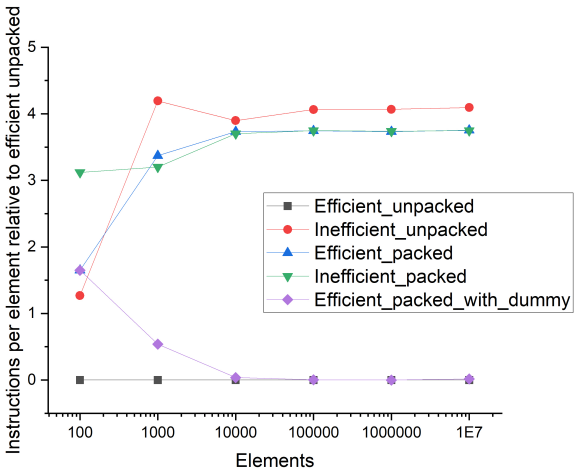
```
{
    Struct_type d;
    d.c = rand_char;
    d.s = rand_short;
    d.i = rand_int;
    d.d = rand_double;
    container.push_back(d);
}
Struct_type d;
for (int j = 0; j < num_elements; j++)
{
    d = container[j];
}
```



(a) Vector without reserve



(b) Vector with reserve



(c) Array

Figure 3.20: Cache loads per element (relative to efficient unpacked) for appending to and reading from a vector/array

Or as follows (for an array):

```
for (int i = 0; i < num_elements; i++)
{
    Struct_type d;
    d.c = rand_char;
    d.s = rand_short;
    d.i = rand_int;
    d.d = rand_double;
    container[i] = d;
}
Struct_type d;
for (int j = 0; j < num_elements; j++)
```

```
{
    d = container[j];
}
```

Figure 3.16 shows the latency data per element with varying numbers of elements. The first and simplest result is that inefficient ordering of data members within structs can have a significant effect on latency. It is fairly obvious why this should be the case: they are larger and therefore require more time to copy and read. Indeed, it is easy to see the difference between the assembly for an efficient struct (for simplicity's sake in the array test):

```
d2 = arr[j];
    12d8:       mov     rax,QWORD PTR [rbp-0x30]
    12dc:       mov     edx,DWORD PTR [rbp-0x3c]
    12df:       movsxd  rdx,edx
    12e2:       shl     rdx,0x4
    12e6:       add     rax,rdx
    12e9:       mov     rdx,QWORD PTR [rax+0x8]
    12ed:       mov     rax,QWORD PTR [rax]
    12f0:       mov     QWORD PTR [rbp-0x20],rax
    12f4:       mov     QWORD PTR [rbp-0x18],rdx
```

and the assembly for an inefficient struct:

```
d2 = arr[j];
    12f2:       mov     rcx,QWORD PTR [rbp-0x30]
    12f6:       mov     eax,DWORD PTR [rbp-0x3c]
    12f9:       movsxd  rdx,eax
    12fc:       mov     rax,rdx
    12ff:       add     rax,rax
    1302:       add     rax,rdx
    1305:       shl     rax,0x3
    1309:       add     rcx,rax
    130c:       mov     rax,QWORD PTR [rcx]
    130f:       mov     rdx,QWORD PTR [rcx+0x8]
    1313:       mov     QWORD PTR [rbp-0x20],rax
    1317:       mov     QWORD PTR [rbp-0x18],rdx
    131b:       mov     rax,QWORD PTR [rcx+0x10]
    131f:       mov     QWORD PTR [rbp-0x10],rax
```

It is interesting to note that a lot of the discrepancy comes from the difference between the process of calculating the offset of an element by multiplying by 16 (lines 12df to 12e6 in the efficient example)) and multiplying by 24 (lines 12f9 to 1309 in the inefficient example). Moving the contents of an element from one place to another also requires three 8-byte quadword moves for the inefficient struct as opposed to two for the efficient struct. Relatedly, it is also interesting to note that the difference between the efficiently ordered struct and the inefficiently ordered struct is much greater when the vector does not reserve space to hold all the elements in advance. When no space is reserved in advance, a lot more copying is required, and

this amplifies the extra instructions (and cache references) that the inefficient struct requires.

The second interesting set of results from these tests relates to the packed structs. As expected (since they occupy the same amount of space and their sub-members will be misaligned roughly the same amount of the time), the efficiently ordered and inefficiently ordered packed structs behave virtually the same in all circumstances. The first thing to note about the packed structs is that they are consistently slower than the efficient unpacked structs. From this it can be concluded that minimising space requirements is not the only factor to consider. However, it is interesting that in the case of a vector which does not reserve space in advance, packing the inefficient struct comes with a significant latency improvement, so space considerations are a factor even if not the only one.

This prompts the question of what it is exactly that makes the packed structs slower than the inefficiently ordered unpacked struct in the other two cases (where the elements are in a vector that reserves space in advance and where the elements are in a plain array). Because the data structures involved in each test are constant and only the structs differ (in ordering and/or packing), it might be useful to look at pure measures of cache efficiency. However, it is quickly clear from Figure 3.17 that this cannot be the source of the slowness of packed structs: packed structs show a better cache hit ratio than the both kinds of unpacked struct. Further, Figure 3.18 clearly demonstrates that the number of instructions per cycle does not match the pattern of latencies.

Given that these 'pure' measures of cache efficiency do not appear to explain the observed latencies, it seems appropriate to take inspiration from the previous section on sequence and associative containers and look to the number of instructions that need executing and the number of cache references that need to be made. A cursory glance at the assembly for assigning to the inefficiently ordered unpacked struct and the packed struct would give the (false) impression that this was indeed the source of the problem. Here is the assembly for the inefficient unpacked struct (as above):

```
d2 = arr[j];
    12f2:       mov     rcx,QWORD PTR [rbp-0x30]
    12f6:       mov     eax,DWORD PTR [rbp-0x3c]
    12f9:       movsxd  rdx,eax
    12fc:       mov     rax,rdx
    12ff:       add     rax,rax
    1302:       add     rax,rdx
    1305:       shl     rax,0x3
    1309:       add     rcx,rax
    130c:       mov     rax,QWORD PTR [rcx]
    130f:       mov     rdx,QWORD PTR [rcx+0x8]
    1313:       mov     QWORD PTR [rbp-0x20],rax
    1317:       mov     QWORD PTR [rbp-0x18],rdx
    131b:       mov     rax,QWORD PTR [rcx+0x10]
    131f:       mov     QWORD PTR [rbp-0x10],rax
```

And here is the assembly for the packed structs:

```
d2 = arr[j];
    12f1:       mov     rcx,QWORD PTR [rbp-0x28]
    12f5:       mov     eax,DWORD PTR [rbp-0x34]
    12f8:       movsxd  rdx,eax
    12fb:       mov     rax,rdx
    12fe:       shl     rax,0x4
    1302:       sub     rax,rdx
    1305:       add     rax,rcx
    1308:       mov     rdx,QWORD PTR [rax]
    130b:       mov     QWORD PTR [rbp-0x17],rdx
    130f:       mov     edx,DWORD PTR [rax+0x8]
    1312:       mov     DWORD PTR [rbp-0xf],edx
    1315:       movzx   edx,WORD PTR [rax+0xc]
    1319:       mov     WORD PTR [rbp-0xb],dx
    131d:       movzx   eax,BYTE PTR [rax+0xe]
    1321:       mov     BYTE PTR [rbp-0x9],al
```

While multiplying by 15 (lines 12f8 to 1305) is quicker than multiplying by 24, the compiler is forced to move each of the sub-elements of the packed struct individually rather than ferrying them all across with two 8-byte quadword moves. However, it would be a mistake to conclude one's dive into the assembly code at this point. In many other places in the code, the inefficient struct leads to many more instructions. For example, the code for the function _Alloc_traits::construct is longer for the inefficient than the packed struct, as is the code for the _M_deallocate function.

More importantly, Figure 3.19 shows empirically that inefficient unpacked structs lead to more instructions per element. Further, Figure 3.20 shows that numbers of cache loads is also not the answer.

So, unlike in the preceding tests, the source of the latency problems with packed structs is not to be found in the numbers of instructions or cache references. Another interesting (but again misleading) result can be seen from the purple 'efficient packed with dummy' line in each of the foregoing figures. The efficient packed struct with a dummy element is as follows:

```
struct Efficient_packed_with_dummy
{
    char c;
    short s;
    int i;
    double d;
    char dummy;
} __attribute__((packed));
```

The dummy element is never assigned and never accessed, it simply forces the struct as a whole onto 16-byte alignments by enlarging it one byte. In almost all cases adding the dummy element eliminates the difference between the packed struct and the inefficient unpacked struct. However, while it may seem tempting to do so, it is not possible to conclude from this that it is alignment on multiples of 15 vs alignment on multiples of 16 that leads to this effect. Unfortunately for the purposes of these

experiments, the compiler is clever enough to realise that, packed though it may be, the packed struct with the dummy element can still be moved with two 8-byte quadword moves, just like the efficient unpacked struct. The following assembly code is from the efficient packed struct with dummy element (in an array):

```
d2 = arr[j];
    12d8:       mov     rax,QWORD PTR [rbp-0x30]
    12dc:       mov     edx,DWORD PTR [rbp-0x3c]
    12df:       movsxd  rdx,edx
    12e2:       shl     rdx,0x4
    12e6:       add     rax,rdx
    12e9:       mov     rdx,QWORD PTR [rax+0x8]
    12ed:       mov     rax,QWORD PTR [rax]
    12f0:       mov     QWORD PTR [rbp-0x20],rax
    12f4:       mov     QWORD PTR [rbp-0x18],rdx
```

The same thing occurs when using the alignas() specifier to force packed elements onto 16-byte alignments.

In short, the question of the effects of alignment on the (slow) behaviour of packed elements is not answered by the (faster) behaviour of packed elements with dummy elements added; this is not a speed-up born of alignment but rather a speed-up born of fewer instructions needing to execute.

However, there are still some extremely interesting results that can be elicited by adding dummy elements to both packed and unpacked structs, thereby controlling more carefully for these confounding factors, and looking at the alignment of the sub-elements of the structs rather than the alignment of the structs as a whole. Specifically, a dummy element can also be added to the efficient unpacked struct in the blank byte between the char and the short as follows:

```
struct Efficient_unpacked_with_dummy
{
    char c;
    char dummy;
    short s;
    int i;
    double d;
};
```

As mentioned above, this kind of struct behaves identically to the efficient unpacked struct without the dummy char in the blank byte in all observed cases. Moreover, with the dummy element in place in the previously blank byte of the unpacked struct, all other things really are equal to the packed struct with the dummy element on the end: the only difference is the alignment of the internal sub-elements of each struct, i.e., the sub-elements of the unpacked struct (specifically the short, the int, and the double) are aligned, while the sub-elements of the packed struct are not aligned.

Looking at the latency data in Figure 3.16 with this in mind, the discrepancy between the pattern shown by the arrays and vectors of 16-byte packed/unpacked structs with dummy elements is interesting for a new reason. To be clear, plain

arrays do not show the same pattern of packed and unpacked structs with dummy elements as vectors do. Specifically, vectors of 16-byte unpacked structs (now with dummy elements) show the same execution latencies as vectors of 16-byte packed structs with dummy elements, while arrays of the former type show lower execution latencies than arrays of the latter type.

With the alignment of the sub-elements of each struct in mind, these two patterns appear to lead to directly contradictory conclusions. The vector data lead one to conclude that the misalignment of the short, int, and double in the packed struct does incur a latency penalty; the array data lead one to conclude that it does not.

Interestingly, there appears to be further evidence that there is a misaligned access penalty in the form of another fascinating result. Putting 16-byte efficient unpacked and packed objects with dummy elements into vectors and arrays at different levels of optimisation using the gcc flags -O0, -O1, -O2, and -O3 produces the results shown in Figure 3.21. Without optimisation, the same pattern as in Figure 3.16 emerges, i.e., unpacked and packed structs with dummy elements behave the same in vectors, but the former are faster than the latter in arrays. When optimised, however, a different pattern emerges: in all cases, there is no apparent misalignment effect.

Why this is evidence for a latency penalty associated with misalignment might not be immediately clear. When one looks at the assembly the compiler produces with optimisation flags, however, it becomes clearer. The following assembly is produced for unpacked structs with a dummy element in a vector without optimisation:

```
d.c = rand_char;
    a86d:       movzx   eax,BYTE PTR [rbp-0xab]
    a874:       mov     BYTE PTR [rbp-0x40],al
d.s = rand_short;
    a877:       movzx   eax,WORD PTR [rbp-0xaa]
    a87e:       mov     WORD PTR [rbp-0x3e],ax
d.i = rand_int;
    a882:       mov     eax,DWORD PTR [rbp-0x9c]
    a888:       mov     DWORD PTR [rbp-0x3c],eax
d.d = rand_double;
    a88b:       movsd   xmm0,QWORD PTR [rbp-0x90]
    a892:
    a893:       movsd   QWORD PTR [rbp-0x38],xmm0
}
```

The following is produced for packed structs with a dummy element in an array without optimisation:

```
d.c = rand_char;
    a86d:       movzx   eax,BYTE PTR [rbp-0xab]
    a874:       mov     BYTE PTR [rbp-0x40],al
d.s = rand_short;
    a877:       movzx   eax,WORD PTR [rbp-0xaa]
    a87e:       mov     WORD PTR [rbp-0x3f],ax
d.i = rand_int;
```

```
    a882:       mov     eax,DWORD PTR [rbp-0x9c]
    a888:       mov     DWORD PTR [rbp-0x3d],eax
d.d = rand_double;
    a88b:       movsd   xmm0,QWORD PTR [rbp-0x90]
    a892:
    a893:       movsd   QWORD PTR [rbp-0x39],xmm0
```

The following, in contrast, is produced for unpacked structs with a dummy element in an array with optimisation:

```
d.d = rand_double;
    a90a:       movsd   xmm0,QWORD PTR [rsp]
if (this->_M_impl._M_finish != this->_M_impl._M_end_of_storage)
    a90f:       mov     rsi,QWORD PTR [rsp+0x38]
d.c = rand_char;
    a914:       mov     BYTE PTR [rsp+0x20],bl
d.s = rand_short;
    a918:       mov     WORD PTR [rsp+0x22],r12w
d.i = rand_int;
    a91e:       mov     DWORD PTR [rsp+0x24],r13d
d.d = rand_double;
    a923:       movsd   QWORD PTR [rsp+0x28],xmm0
```

And the following for packed structs with a dummy element in an array with optimisation:

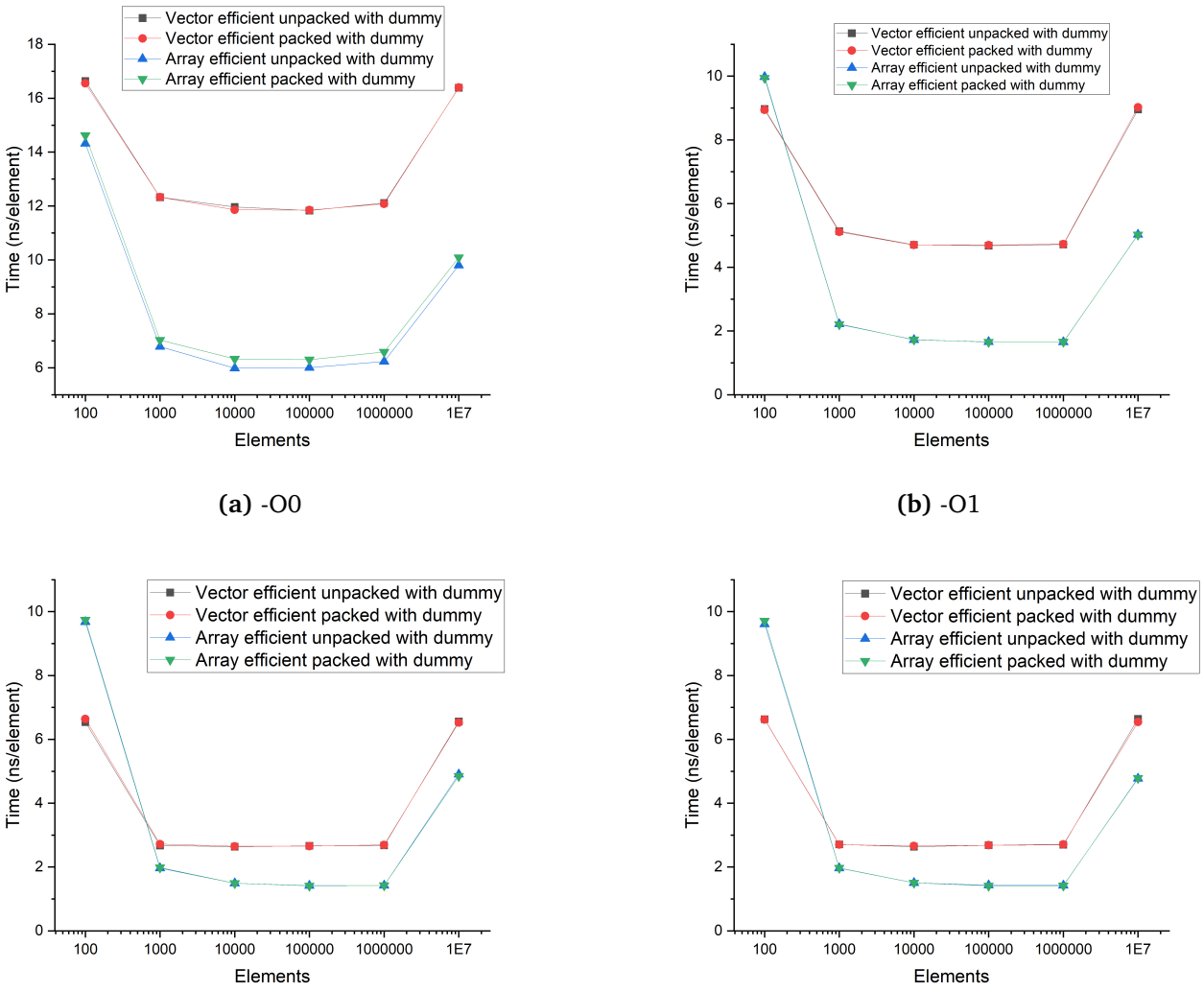
```
d.d = rand_double;
    a90a:       movsd   xmm0,QWORD PTR [rsp]
if (this->_M_impl._M_finish != this->_M_impl._M_end_of_storage)
    a90f:       mov     rsi,QWORD PTR [rsp+0x38]
d.c = rand_char;
    a914:       mov     BYTE PTR [rsp+0x20],bl
d.s = rand_short;
    a918:       mov     WORD PTR [rsp+0x21],r12w
d.i = rand_int;
    a91e:       mov     DWORD PTR [rsp+0x23],r13d
d.d = rand_double;
    a923:       movsd   QWORD PTR [rsp+0x27],xmm0
```

The critical difference is that, in the unoptimised version, rand_char, rand_short, rand_int, and rand_double are stored on the stack (i.e., in cache memory), so each assignment to a new element requires many references to caches (both reads and writes), while in the optimised version, these elements are stored in registers, so each assignment of a new element requires fewer references to caches. Since reduced references to caches eliminates the difference between unpacked and packed elements (the alignment and unalignment of their respective elements can be seen from the referenced addresses in the above assembly), this is evidence that it is in these cache references that the latency penalty originates, lending credence to the theory that unaligned accesses incur a latency penalty over aligned accesses.

In fact, it is possible to draw more specific conclusions from these results. The unoptimised code requires many reads and many writes from cache memory, while the optimised code requires just as many writes but far fewer reads. As such, it appears that reads from unaligned memory locations incur a misalignment penalty, but writes do not. This possible conclusion certainly merits further research.

Note that although this is evidence that there is a misaligned memory access penalty, it does not provide an explanation for vectors not showing a misaligned access penalty in Figure 3.16. It remains to be shown why this is the case, but I can propose a potential explanation: the extra instructions involved in the operator[] and elsewhere in the unoptimised vector code that aren't present in the array code combined with processor pipelining may give the misaligned access penalty a 'place to hide'; that is, the (small) access penalty may be lost in the time that the pipelined data-fetching instructions spend waiting to arrive in the processing unit.

While this tentative proposal also merits further research, taken together with the optimisation data above it provides a way to explain both why (a) vectors do not show a difference between the two types of 16-byte struct, and (b) arrays do show a difference.



(a) -O0 **(b)** -O1

(c) -O2 **(d)** -O3

Figure 3.21: Latencies per unpacked/packed element for appending to and reading from a vector/array at different levels of optimisation

Interim conclusions Following on closely from the foregoing section, one thing it is certainly possible to conclude from these findings is that researchers must be unwaveringly fastidious about controlling for interfering factors when drawing conclusions from their experiments pertaining to data packing and misaligned accesses. Many authors simply describe the latency data collected from executing their code. It is clear, however, that investigating the factors that underlie these latencies (instructions, cache loads, cache misses, and so on) is vital, as is looking carefully at the compiled assembly and the decisions that compilers make that may not be immediately apparent to a C++ programmer.

Returning to the questions posed at the start of this section, these tests suggest some answers. First, it is imperative that data members in structs are ordered as efficiently as possible, as inefficient orderings result in significant latency penalties. When specific complex access patterns are taken into account, the question of data

organisation and management becomes a bit more complicated; it has been pointed out that it is more efficient to take members that are accessed across multiple objects one after the other out of those objects and to put them into dedicated arrays of their own in order to avoid large access strides and favour contiguous consecutive accesses (i.e., to create 'structs of arrays' rather than 'arrays of structs' [5, 55]). However, these considerations aside, efficient ordering of elements in structs has been shown to make a significant difference to speed of execution.

Further, data packing generally appears to be disadvantageous. If it is guaranteed that there will be a lot of data copying in large contiguous chunks, a smaller packed structure is better than an inefficiently ordered unpacked one; in all other cases observed here, packed data suffers in comparison with efficiently ordered unpacked data, most likely (according to the results I have collected) as a result of misalignment of sub-elements. These findings, then, side with some authors [18] and against some others [20, 21]. It is of course the case that the extent of the penalties for misaligned accesses may differ processor to processor, and it may well be the case that some modern processors have eliminated misaligned access penalties altogether [48]. However, the processors that these tests were run on are not very old and appear to show evidence of misaligned-access penalties. However, it would be highly worthwhile to run these tests on more processors and to answer this question more definitively with more data.

3.5 Summary of conclusions

The foregoing investigation has presented a large number of tests in a significant amount of detail, and leads to several interesting conclusions.

From Section 3.2 it can be concluded that programmers aiming to improve cache-efficiency for latencies on the order of micro- and nano-seconds cannot afford to ignore any dimension of their data- or instruction-caches. In order to achieve maximum cache-efficiency, as few regions of data and of code must compete for cache lines as possible. Even below the associativity of these caches jitter can be a problem, and in the context of HFT it is necessary to minimise this measure as well as minimising averaged latencies.

From Section 3.3 it can be concluded that contiguous data structures which do not involve frequent memory allocation and pointer dereferencing should be avoided in favour of contiguous data structures. However, rather than being less 'cache efficient' in the naive sense that some authors imply (i.e., that discontiguous data structures result in a higher cache miss percentage and lower instructions executed per cycle), the cache-inefficiency of these discontiguous structures actually manifests itself in higher absolute numbers of instructions executed and higher absolute numbers of cache loads, and this is the source of the offending latency discrepancy. However, it is necessary to consider how these data structures handle large amounts of data. Contiguous data structures are sometimes forced to resize and therefore reallocate, which involves a lot of copying. When the number of elements in each data structure is large, the contiguous data structures require large amounts of copying and therefore become slower than the discontiguous data structures.

Finally, from Section 3.4 several further conclusions can be drawn and a number of issues remain outstanding. It is certainly the case that data ordering in user-defined types matters, in that inefficiently ordered types result in significantly higher latencies than efficiently ordered types, especially when a lot of copying is involved. However, it does not appear to be the case that packing efficiently ordered types leads to faster execution latencies; rather, the misalignment of sub-elements of packed structs appears to lead to latency slow-downs. More research is needed on this latter point, however.

Chapter 4

Software contribution

In order to apply the foregoing research in the interests of actual latency improvements, I have developed a tool for detecting data- and instruction-cache inefficiencies. The tool is invoked from the command line, and looks at user-provided source-code files and compiled binary files to detect data- and code-inefficiencies and suggest improvements to the programmer. Importantly, programmers writing for ultra-low-latency applications such as High Frequency Trading need to programme with the specific cache on which their code will run in mind; code that avoids jitter on one processor may not avoid it on another. As such, the tool that I have developed performs cache analysis as well as code and binary analysis. There are many ways in which the user can configure the tests and analysis that the tool performs, and I will discuss these in the following.

4.1 Outline and general use

The tool performs three kinds of analysis — cache analysis, source code analysis, and binary analysis — and integrates findings from all three of these analyses to detect inefficiencies in data (i.e., user-defined types) and code (i.e., functions). The data-inefficiencies that the tool detects are user-defined types with inefficient data-member orderings (cf. Section 3.4) and user-defined types whose total size has a small Least Common Multiple with the critical stride of their processor's cache, leading to potential cache-conflicts between corresponding data members in contiguously allocated objects of that type (cf. Section 3.2). The code-inefficiencies that the tool detects are large functions (those which are larger than the critical stride of the processor, meaning that they may compete with themselves for cache space), and groups of functions which both call each other (which 'coexecute') and compete with each other for cache space (cf. Section 3.2).

The tool is ordinarily invoked on the command line with a list of files in the following format:

./latency_tool [–options] [source_files ...] binary_file

If no specific options are chosen, the tools prints out an analysis of the processor's

cache structure, analyses the specified source-code files and binary files, flags up data-cache inefficiencies to the user, suggests solutions to these inefficiencies, flags up instruction-cache inefficiencies to the user, and suggests solutions to these inefficiencies likewise.

As far as I can find, no other tool exists which performs this kind of analysis and provides this kind of functionality. As discussed in Section 2.4, I have built on several disparate tools which provide the capability to perform these cross-source analyses (nm, objdump, gdb) and there exist certain tools which perform optimisations opaquely ('in the background'), but no tool is available which identifies these kinds of cache inefficiencies and reports them to the user alongside suggestions for how to mitigate them. Of course, it follows that there is also no existing tool which integrates data and code analysis with cache structure analysis, and no existing tool which suggests cache efficiency improvements to programmers tailored to their specific code and their specific processor.

In the following I will discuss the internal workings and functionality of the tool I have developed. I will describe its initial setup (input parsing, validity checks, and the like) in Section 4.2, how it performs its analysis of the cache in Section 4.3, how it performs its data efficiency analysis in Section 4.4, and how it performs its code efficiency analysis in Section 4.5. I will describe the configurations and customisations available to the user in these sections where they are relevant, and finish with other features in Section 4.6.

4.2 Programme setup

The first thing the programme does is parse the user's chosen options flags. The results of this input determine what constitutes valid command line input, and which dependencies need to be present on the user's system in order for the programme to the provide the desired functionality. For example, while normal use involves specifying source code and binary files, the option is available to the user to have the tool simply print out its analysis of their processor's cache structure (with the flag -c or –cache-info-only); with this option, an empty list of files is acceptable. The results of these options are stored in a UserOptions object, and the values of binary options are stored efficiently in a std::bitset so that their values can be interrogated with the std::bitset::test function. For readability and configurability, the bit-positions of the various options in the bitset are written into the code using preprocessor #define directives.

After parsing the user's chosen options, the programme checks if all files specified by the user do in fact exist on the system. Checking this early in the process avoids wasting any processing time and prevents any problems that may arise from attempting to read non-existent files.

Next, the programme checks whether the required dependencies are present on the user's system. When using the tool normally, these dependencies are: g++ [56, 57], gdb [22], perf [27], nm [25, 26], and objdump [23, 24], all of which are common and easily sourced utilities. Which of these are required in specific use cases is dependent on the options specified by the user. For example, if the user only

wants information about their cache information, gdb, nm, and objdump are not required; if the user specifies that no empirical cache tests should be performed (cf. Section 4.3), g++ and perf are not required.

4.3 Cache analysis

Analysis of the user's processor and its cache structure is integral to providing tailored suggestions. (At present, the programme only uses data pertaining to the L1 data- and instruction-caches. However, the framework is in place in the code to extend this to analyses pertaining to lower level caches straightforwardly.) The goal of the cache analysis part of the programme is to ascertain the cache dimensions that are to be used to identify potential sources of data- and instruction-cache inefficiencies and to facilitate the provision of tailored suggestions to the user.

If no options are chosen that indicate otherwise, the first thing the programme does to acquire information about the processor's cache structure is to query the system with the following function calls:

```
sysconf(_SC_LEVEL1_DCACHE_SIZE)
sysconf(_SC_LEVEL1_DCACHE_LINESIZE)
sysconf(_SC_LEVEL1_DCACHE_ASSOC)
sysconf(_SC_LEVEL1_ICACHE_SIZE)
sysconf(_SC_LEVEL1_ICACHE_LINESIZE)
sysconf(_SC_LEVEL1_ICACHE_ASSOC)
sysconf(_SC_LEVEL2_CACHE_SIZE)
sysconf(_SC_LEVEL2_CACHE_LINESIZE)
sysconf(_SC_LEVEL2_CACHE_ASSOC)
sysconf(_SC_LEVEL3_CACHE_SIZE)
sysconf(_SC_LEVEL3_CACHE_LINESIZE)
sysconf(_SC_LEVEL3_CACHE_ASSOC)
sysconf(_SC_LEVEL4_CACHE_SIZE)
sysconf(_SC_LEVEL4_CACHE_LINESIZE)
sysconf(_SC_LEVEL4_CACHE_ASSOC)
```

It is of course not always the case that the programmer wishes to identify inefficiencies in the context of the same processor they are using to run the tool. As such, the user has the option to provide a different cache specification and ignore the cache on the processor they're running the tool on by using the following flag:

-m=⟨size⟩:⟨associativity⟩:⟨linesize⟩::⟨size⟩:⟨associativity⟩:⟨linesize⟩::...

(or –manual-cache=...)

The caches are specified in the order: L1 data, L1 instruction, L2, L3, and L4. Any number of these caches can be provided; caches that are not provided are ignored. Further, instead of providing numerical dimensions for all the relevant caches, the user can simply use the flag with =default, which will choose default values of

32768-byte L1 data- and instruction-cache sizes, 64-byte linesizes, and 8-way associativities. These default values were chosen by examining many machines and taking the modal values.

On some systems (it is not clear to me exactly why certain systems behave this way), some but not all of these system variables have values. For example, on my own desktop computer at home, the L1 instruction cache associativity system variable is simply unspecified. The following command run on that particular machine gives the following output:

```
getconf -a | grep CACHE

LEVEL1_ICACHE_SIZE         32768
LEVEL1_ICACHE_ASSOC
LEVEL1_ICACHE_LINESIZE     64
LEVEL1_DCACHE_SIZE         32768
LEVEL1_DCACHE_ASSOC        8
LEVEL1_DCACHE_LINESIZE     64
LEVEL2_CACHE_SIZE          262144
LEVEL2_CACHE_ASSOC         8
LEVEL2_CACHE_LINESIZE      64
LEVEL3_CACHE_SIZE          8388608
LEVEL3_CACHE_ASSOC         16
LEVEL3_CACHE_LINESIZE      64
LEVEL4_CACHE_SIZE          0
LEVEL4_CACHE_ASSOC
LEVEL4_CACHE_LINESIZE
```

When the associativity of a cache is unknown, its critical stride cannot be calculated. Without these two values, latency/jitter inefficiencies that arise from the structure of the cache cannot be identified. As such, if the user does not choose any options specifying otherwise, the tool will proceed to analyse the processor it is running on empirically. This functionality takes directly from the analysis in the foregoing data section of this project (cf. esp. Section 3.2). The empirical tests of the processor proceed as follows.

Data cache associativity tests If the data cache does not specify its associativity, the tool runs the following compilation command on a source code file which comes provided with the tool:

```
g++ -g ./test_files/associativity_test_d.cpp -o ./temp_files/associativity_test_d
```

Then, the programme runs the following benchmark command several times with ‘assoc’ cycled through from 2 to 16, and ‘stride’ set to 16 (wherein the data is not on aligned memory addresses) and 65536 (wherein the data will be on aligned memory addresses):

perf stat -x , –append -o ./temp_files/assoctmpd.txt -e L1-dcache-load-misses -r 1000 ./temp_files/associativity_test_d ⟨assoc⟩ ⟨stride⟩

The cache miss percentages and standard deviations in cache miss percentages are then processed in order to make an informed estimate of the associativity of the data cache. The calculation with the widest applicability across systems is to take the earliest (i.e., the smallest implied associativity) of (a) the first jump in cache miss percentage of 65536-byte-aligned data above the 16-byte-aligned data and (b) the highest standard deviation in cache miss percentage (cf. Figure 3.2).

Data cache critical stride tests After forming an estimate of the data cache associativity, the tool then runs the following compilation and benchmark commands, cycling 'stride' through the powers of two from 64 to 65536:

g++ -g ./test_files/critical_stride_test_d.cpp -o ./temp_files/critical_stride_test_d

perf stat -x , –append -o ./temp_files/cstmpd.txt -e L1-dcache-loads,L1-dcache-load-misses,duration_time -r 1000 ./temp_files/critical_stride_test_d ⟨stride⟩

The best indication of the critical stride is the standard deviation in cache miss percentage: the highest value is found at a quarter of the critical stride (cf. Figure 3.1).

If the estimated values of the data-cache associativity and critical stride are in agreement (i.e., the size divided by the estimated associativity is the same as the estimated critical stride), these values are stored and used in the ensuing analysis. If they do not agree, first the programme checks if one suggests a power-of-two critical stride and the other suggests a non-power-of-two critical stride. If so, the power-of-two is assumed to be correct. If both or neither are powers of two, the smaller of the two suggested critical strides is chosen, because an incorrect but smaller critical stride will give rise to extra suggestions that will subsume all the pertinent ones, whereas an incorrect but larger critical stride will not do so. This cross-checking of the critical stride and associativity values adds robustness to the estimates that the empirical tests generate.

Instruction cache associativity tests The empirical instruction-cache associativity tests are similar to the corresponding tests on the data-cache. However, instead of passing data alignments as command line arguments, comparing aligned and non-aligned functions requires separate compilations with the following commands using the gcc flag –falign-functions:

g++ -g -falign-functions=65536 ./test_files/associativity_test_i.cpp -o ./temp_files/associativity_test_i_65536

g++ -g ./test_files/associativity_test_i.cpp -o ./temp_files/associativity_test_i_noalign

The programme then runs the following two benchmark commands several times each, varying the values of 'number_of_functions':

perf stat -x , –append -o ./temp_files/assoctmpi.txt -e L1-icache-load-misses -r 1000 ./temp_files/associativity_test_i_65536 ⟨number_of_functions⟩

perf stat -x , –append -o ./temp_files/assoctmpi.txt -e L1-icache-load-misses -r 1000 ./temp_files/associativity_test_i_noalign ⟨number_of_functions⟩

Using the data produced by these benchmarks, the best indication of the associativity of the instruction cache is the same as the best indication for the data-cache (cf. Figures 3.2 and 3.3).

Instruction cache critical stride tests Finally, the instruction-cache critical stride tests also involve multiple compilations, using the following command with 'alignment' cycled through the powers of two from 64 to 65536:

g++ -g -falign-functions=⟨alignment⟩ ./test_files/critical_stride_test_i.cpp -o ./temp_files/critical_stride_test_i_⟨alignment⟩

The following benchmark command is then run, cycling 'alignment' through the same values:

perf stat -x , –append -o ./temp_files/cstmpi.txt -e L1-icache-load-misses -r 1000 ./temp_files/critical_stride_test_i_⟨alignment⟩

Using the data from these benchmarks, the best indicator of the critical stride of the instruction-cache is the last significant jump in the number of instruction-cache misses (cf. Figure 3.3).

Suppressing empirical tests There are numerous reasons why a user might not want the tool to run these empirical tests even when certain system variables relating to cache dimensions are missing. For example, they may not want to wait the minute or two that the tests take, or they might not have perf or g++ installed on their system (or their use might be restricted by an administrator!). As such, the user has two related options which suppress these empirical tests in the case that the system variables do not provide the requisite information.

The first option is to use the following command line flag:

-n/–no-empirical-tests

If this option is chosen and the system variables are incomplete, default values are chosen and the user is warned that default values will be used. Again, the default values were chosen by simply checking many machines and looking at the modal values. The default critical stride is 4096 bytes, the default associativity 8, and the default linesize 64 bytes.

Another reason the user might not want to run empirical tests is that they ran empirical tests last time they used the tool and do not want the same tests to be run again. In this case the user can use the following command line flag:

-e/–use-existing-cache-temp-files

If this option is chosen, the programme looks for existing temp files containing data produced by the above benchmark commands. If some files are missing, the programme warns the user and terminates. This flag also automatically switches on the -k/–keep-temp-files flag, which suppresses the automatic deletion of temp files that occurs when the programme terminates.

4.4 Data-efficiency analysis

The goal of the data-efficiency analysis functionality of this tool is to identify sources of data-cache inefficiency and suggest solutions to these problems to the user. The overall approach this tool takes is to identify user-defined types, to identify the overall size of those types, to identify the data members of those types, and to identify the sizes and alignments of those data members. This information can then be used to identify inefficient type sizes and inefficient data member orderings.

Attempt 1: writing a C++ class parser My first attempt at engineering this functionality was to manually ascertain all the relevant information from the source code alone. I spent a lot of time working on this and got a fairly long way with it. Indeed, for many cases, it was able to provide all the desired functionality of identifying classes, their total size, their data members, the sizes of their data members, diagnosing inefficient orderings, and suggesting optimised orderings. However, eventually, the number of challenges I uncovered became insurmountable, and I abandoned this for another method. However, it was an interesting undertaking and is therefore worth describing in more detail.

The first step was to identify user-defined types (classes and structs). The following regular expression identifies the beginning of a class and accounts for the highly fluid spacing of legal C++ code and potential templating:

”(class| struct)\\s*([a-zA-Z_][a-zA-Z0-9_]*)\\s*(:\\s*(public| protected | private)\\s*([a-zA-Z_][a-zA-Z0-9_⟨⟩]*)\\s*)?\\{”

It is well known that the regular expressions, which are as powerful as the class of regular languages, are unable to do recursion because recursion requires counting and counting is impossible under the constraints of regular languages (but not

context-free grammars) [58]. Since classes are delimited by curly brackets and legal C++ code can recursively contain more levels of curly brackets, specific functionality had to be included to identify the curly bracket (along with optional variable name and obligatory semicolon) that ends each class and to distinguish that curly bracket from other curly brackets within the class (which may also be followed, within the class, by variable names and semicolons).

Once each class was identified, the next stage was to identify the data members of the class. Many challenges arose in this step, a lot of which were overcome and some of which were not. One challenge was user types which contain other user types which may not yet have been processed by the parser and which may be in other files. This problem was amplified by the problem of calculating the sizes of user types containing other user types as members, which I will discuss more below. Another text-parsing problem was ignoring text that would constitute valid variable declarations but is in speech marks, slash-slash comments, or slash-star comments. Additionally, function parameters of the following kind look exactly like data member declarations:

```
int func(int a = 4, char g, short* s = nullptr) { ... }
```

Variable declarations inside functions also look like valid data member declarations. Moreover, pure virtual function declarations also look a lot like member variable declarations:

```
virtual int not_a_data_member = 0;
```

All of these problems were overcome. For example, the nested class problem was solved by reading in all the classes in all the source code files, identifying their names, and adding them into the data-member declaration regular expression (see below) before moving on to parsing the data members of each class. To solve the comments problem, the parser would keep track of whether it was inside comments or not using Boolean variables and would look out for relevant comment indicators. To solve the function parameter problem, anything inside round brackets was ignored. To solve the function variable problem, anything nested two-or-more curly brackets deep was ignored. To solve the virtual functions problem, virtual functions were actually accepted as valid data members by the regular expression (which was easier than discarding them as invalid) and discarded further down the line by looking for the 'virtual' keyword.

However, several such problems remained unsolved by the time I abandoned this sub-project. For example, my final attempt at identifying data members could not capture comma-separated variable declarations such as the following:

```
int a, b, c = 4, d = 3, e;
```

Nor could it correctly identify templated parameters such as the following:

```
template <typename TPName>
class ClassName
{
    TPName templated_variable;
}
```

When it came to calculating the sizes of data members and the size of the class as a whole, many more challenges arose. Again, many were overcome and some were not. For example, one challenge was identifying whether pointers were inside template parameters or not. More specifically, in the following two examples, the former is (generally) 24 bytes large and the latter (generally) 8 bytes large:

```
std::vector<std::string*>
std::vector<std::string>*
```

Consider also the complexities involved in calculating the sizes of the following:

```
char const * const * c
char a[5]
char a[4 + 1]

std::vector<char[5]> v
std::vector<char>[5] v
std::vector<int> v
std::vector<bool> v
std::array<std::array<char[5], 6>, 7>[3] a

#define NUM 4
char a[NUM + 1]

enum Direction { UP, DOWN = 3, SIDEWAYS, IN };
char a[SIDEWAYS + 1]
```

Indeed, it is probably the case that many of these declarations could not be properly interpreted without writing an entire preprocessor. Further problems with calculating the sizes of classes abound. Static variables are not included in the size of a class; derived classes inherit data members and therefore size from their base class(es); classes containing virtual functions involve a hidden vtable pointer which adds 8 bytes to the size of the class.

Compounding this problem is the fact that that, in order to calculate the size of a class as a whole, it is necessary to calculate the alignment of each data member, which is often not the same as its size. For example, the example above with nested std::arrays aligns by the alignment of char, which is 1 byte.

Again, these problems were overcome in various ways, often by inelegantly catching special cases. However, as above, at the point when I moved on to other solutions there were problems with calculating sizes and alignments that were unsolved. For example, the parser could not calculate the sizes of templated classes.

Attempt 2: using llvm-dwarfdump The second approach to calculating the sizes of classes and their data members was to use llvm-dwarfdump [59] to parse the debugging symbols in the compiled binary and print out relevant information. For example, llvm-dwarfdump prints out class information in the following format:

0x000143a0: DW_TAG_class_type
DW_AT_name (”FileCollection”)
DW_AT_byte_size (0x30)
DW_AT_decl_file (”/homes/apb121/latency_tool/data_analyser.hpp”)
DW_AT_decl_line (65)
DW_AT_decl_column (0x07)
DW_AT_sibling (0x00014488)

0x000143ad: DW_TAG_member
DW_AT_name (”file_names”)
DW_AT_decl_file (”/homes/apb121/latency_tool/data_analyser.hpp”)
DW_AT_decl_line (67)
DW_AT_decl_column (0x1e)
DW_AT_type (0x0000c2eb ”vector⟨std::__cxx11::basic_string...[etc. etc.]
DW_AT_data_member_location (0x00)

0x000143ba: DW_TAG_member
DW_AT_name (”udtypes”)
DW_AT_decl_file (”/homes/apb121/latency_tool/data_analyser.hpp”)
DW_AT_decl_line (68)
DW_AT_decl_column (0x19)
DW_AT_type (0x0000d55d ”vector⟨UDType, std::allocator⟨UDType⟩ ⟩”)
DW_AT_data_member_location (0x18)

Using this tool has many advantages, and many of the problems encountered above were no longer a hindrance. However, there were still many problems with this approach. For example, there is still some indirection with regard to identifying types and their sizes, and a lot of complex parsing is still required: nested user-defined types, typedefs, and the like still require looking at some other part of the file to ascertain their size, and alignments still need to be calculated. However, the biggest problem was the sheer size of the dwarfdump file and the prohibitively long time it took to generate and process at runtime: the llvm-dwarfdump of the binary of this software itself, for example, is 741,210 lines long.

Attempt 3 (third time lucky!): using gdb Despite running into problems with the llvm-dwarfdump approach, using debugging symbols in the binary was still a good idea. gdb provides a functionality which simply prints out the size of a specified class and the sizes and offsets of each member of that class. This solves a lot of the problems listed above, but the procedure of ensuring that all user-defined classes (including templated classes) are captured and analysed correctly is still not trivial.

First, it is still necessary to parse the source code directly, at least to identify the names of user-defined types. This is necessary in order to successfully disregard, for example, types defined in external libraries imported in the source code. Having gathered a list of user-defined typenames using the regular expression described above, the tool creates a text file in the following format:

info types ^user_defined_type_1$
info types ^user_defined_type_1⟨.*⟩$
info types ^user_defined_type_2$
info types ^user_defined_type_2⟨.*⟩$
info types ^user_defined_type_3$
info types ^user_defined_type_3⟨.*⟩$
...(etc.)...

The 'info types ⟨regex⟩' gdb command prints out all typenames in the binary that match the regular expression, and, as such, this text file contains commands which guarantee to capture all user defined typenames, correctly including instantiated templated types and ignoring other types. With the above file in place, the following command is run by the programme:

gdb -x ./temp_files/gdbinfocmdtmp.txt -batch ⟨binary_file⟩ ⟩ ./temp_files/
gdbinfoouttmp.txt

This command prompts gdb to execute the list of commands in the specified text file in the context of the specified binary file. gdb then sends the output of those commands to a new file. The resulting file lists the names of all user defined types, including instantiated templated types.

The next step of the process is to generate another text file, this time in the following format, based on the output of the 'info types' commands above:

ptype /o user_defined_type_1
ptype /o user_defined_type_2
ptype /o user_defined_type_3
ptype /o user_defined_type_4
...(etc.)...

The ptype /o gdb command prints out type information and specifies data member offsets (cf. the /o flag). As before, the following command is run to process this text file and produce yet another:

gdb -x ./temp_files/gdbtypecmdtmp.txt -batch ⟨binary_file⟩ ⟩ ./temp_files/
gdbtypeouttmp.txt

This time, the file that gdb produces contains class size and data member information. The programme then parses this file and builds the desired representation of the user-defined types found in the provided source code files.

Diagnosing inefficient data member orderings In order to diagnose inefficient data member orderings, there is still some processing left to do once the gdb output has been parsed. While the size of the class in its current state is known, the total size of proposed new orderings of its data members requires calculating alignments based

on their types, which is not a trivial task and has to be solved in a fairly inelegant, type-by-type fashion. Many of the problems encountered in earlier attempts arose again, but having crafted the foregoing gdb commands carefully, the problems that were left unsolved before were no longer a problem. For example, templated types (and other complex types) appear in their preprocessed form, and comma-separated variables are reformatted into distinct declarations.

Finally, actually calculating the optimal ordering of the data members is perhaps the easiest part of the whole endeavour. Because of the way sizes and alignments work, it is not necessary to try every ordering and see if any of them reduce the size of the class; rather, a greedy algorithm which simply sorts the data members by size (increasing or decreasing) finds the optimally sized ordering. If this optimal size is smaller than the current size, then the user is informed that that type is inefficiently ordered, and the new ordering is printed as an optimisation suggestion.

Diagnosing inefficient class sizes As well as having inefficient orderings, classes may have inefficient sizes. If, for example, a class's size is half the size of the critical stride of the data cache, corresponding data members in every other object in contiguous memory will map to the same cache set. If the class's size is a quarter the size of the critical stride of the data cache, corresponding data members four objects apart will map to the same cache set. If access patterns to contiguously allocated objects of this class involve accessing the same data members in many different objects, this can cause latency problems because the data members in different objects will evict each other from the cache [60]. If there were no such relation between the size of the class and the critical stride of the data cache, the chance of these (common) access patterns leading to cache evictions would be much lower.

This tool identifies potential problems of this kind by calculating the Least Common Multiple (LCM) of the size of each class with the critical stride of the data cache. If the LCM is a small multiple of the size of the class, the tool alerts the user to the potential problem, including details such as how many objects of that class in contiguous memory would lead to a risk of cache evictions. The user is also provided with suggestions for mitigating such problems, such as attempting to make the object smaller, using different data layouts to avoid problematic access patterns ('structs of arrays' rather than 'arrays of structs' [5, 55]), and potentially even making the object slightly larger.

4.5 Code-efficiency analysis

The goal of the code-efficiency analysis part of this tool is to identify instruction-cache inefficiencies. The overall approach that the programme takes is to analyse the compiled binary, identifying function locations and sizes using nm, and identifying functions that call each other by disassembling the binary with objdump and looking for 'call' instructions. This information is then integrated to identify groups of functions which both call each other and which compete for cache space, and inform the user of these groups so they can try to reorganise their code mitigate the problems arising from cache evictions.

Identifying functions, their locations, and their sizes The first step that the tool takes is to use nm to identify user-defined functions. The following command is run on the user-specified binary file:

nm -v -C -l –radix=d –print-size ⟨binary_file⟩ ⟩ ./temp_files/nmtmp.txt

The -v flag orders the output by address rather than name, the -C flag demangles function names, the -l flag includes file names and line numbers from the source code, –radix=d sets the radix to decimal, and –print-size prints the size of each function. User defined functions can be distinguished from other functions by taking all symbols in the 'text section' (code section) of the binary (indicated by nm with the letter 'T') and looking at weak symbols (indicated with the letter 'W') that begin with a user-defined class name (carried over from the data-efficiency analysis described above) and the C++ scope resolution operator '::'.

If the user has not specified any source code files (i.e., has only specified a binary file), it is not possible to distinguish user-defined weak symbols in the binary from, for example, those included from external libraries. The ordinary behaviour of the programme in this circumstance is simply to look at those functions that are in the text section and ignore all weak symbols. However, the user has the option to use the following flag:

-a/–all-functions

This option instructs the tool to consider all functions it finds in the binary. If the user selects this option, they are presented with the warning that, unless the binary is small, this functionality can lead to extremely long run-times.

The programme uses the output from nm and the options chosen by the user to build a representation of the user-defined functions in the binary that contains their names, addresses, sizes, and locations in the source code.

With just this information, it is already possible to identify inefficiently sized functions. If the tool detects a function that is larger than the critical stride of the instruction cache, it prints a warning to the user that details the name of the function, how many times larger than the critical stride it is, and, correspondingly, how many times it might compete with itself for cache space.

Identifying competing functions The first step towards identifying groups of functions that may cause latency problems is to identify which functions compete with which other functions for cache space. In order to do this, the programme uses the base address of each function and its size to work out which cache sets it maps to. Having done this, it compares it to each other function to see if they compete for any cache sets, and thus forms a kind of adjacency list for each function.

Identifying coexecuting functions In order to identify functions which coexecute, the tool employs objdump to disassemble the binary with the following command:

objdump -d -C -Mintel –no-show-raw-insn ⟨binary_file⟩ ⟩ ./temp_files/objdumptmp.txt

The programme then does some assembly language parsing, looking through each user-defined function for 'call' instructions that point to other user-defined functions. These relationships are stored in another set of 'adjacency' lists, which represent, for each user-defined function, which other user-defined functions it calls (and therefore 'coexecutes' with).

There are numerous advantages to using objdump to identify coexecution relationships between functions rather than using the source code files. For example, there are many function calls that aren't explicit in the source code, such as calls to implicitly- (compiler-) defined constructors, destructors, assignment operators, and the like. Further, functions may be called one after the other (and therefore be deemed to coexecute) in a function that is in the source code files but is never actually called by any other function; such functions are not compiled into the binary and so these apparent but misleading coexecutions are correctly ignored. Moreover, two function calls that look identical in the source code may call templated functions with different template arguments which appear in the compiled code as distinct functions. Using the disassembled binary therefore allows the identification of functions and groups of functions that may be invisible from the perspective of the source code — it is possible that implicitly defined constructors, destructors, and two versions of the same templated functions may compete for cache space and cause latency problems — and correctly ignores certain functions and groups of functions that may mistakenly be identified as a source of latency problems.

Identifying problematic groups of functions Having built these representations of which functions compete with which other functions for cache space and which functions call which other functions, the programme performs a set-intersection to form a representation for each function of which functions it both coexecutes with and competes with for cache space. After performing this intersection, it performs a depth first search for coexecuting and competing groups of functions with increasing depth limits (group sizes) until the search finds no groups of that size.

When identifying these groups, the programme keeps track, for each group, of how many times the functions in the group call each other, how much cache space they compete with each other for, and the overall size of the group. Having identified such a group of functions, these three measures are combined into a 'group score' which is used to rank the groups in order of problematic-ness. The tool proceeds to print out the groups as lists of function-names to the user, ordered by their their scores.

If any groups of functions are identified, the tool prints out suggestions for mitigating problems that may arise as a result of these functions coexecuting and competing for cache space. One obvious suggestion is to restructure the code internal to each function to make it smaller, reducing the risk that it competes with the other functions in the group for cache space. Another suggestion is that the user inlines smaller functions into larger functions that they compete with for cache space; if the

smaller function is inside the larger function, it cannot (by definition) compete with it for cache space. Relatedly, a user can also try to combine two or more functions into fewer larger functions. A fourth suggestion is that the user move the functions around in the source code; functions are ordinarily situated in a compiled binary in roughly the same order they appear in the source code, and so moving functions that call each other close to each other in the source code can minimise the risk that they will compete for cache space (cf. [13, 15, 49]). A fifth suggestion is that the user prompt their compiler to force functions onto particular alignments (such as by using the gcc flag -falign-functions=). None of these suggestions is absolutely guaranteed to improve instruction-cache efficiency because each of them may have knock-on effects on the rest of the code, but it is the case that reducing the number of groups of functions that evict each other from the cache will, all other things being equal, improve cache efficiency and execution latencies, and the user can of course use this tool to identify whether modifications they make to their code have this advantageous effect.

User configurability There are many steps in the process of identifying groups of functions that coexecute and compete for cache space described above. Correspondingly, the tool contains many user-configuration options to tailor the analysis to their particular requirements. For example, it may sometimes be too restrictive to simply look at functions that call each other directly, and the user can specify different levels of 'coexecution indirection'. To illustrate, consider functions A(), B(), C(), and D(), where A calls B, B calls C, and C calls D. At zero levels of coexecution indirection, A is considered to coexecute with B only. At one level of coexecution indirection, A is also considered to coexecute with functions that the functions it coexecutes with coexecute with, and would therefore be considered to coexecute with C as well as with B. Likewise, at two levels of coexecution indirection A would also be considered to coexecute with D. The following flag can be used to specify different levels of coexecution indirection:

-l=/–coexecution-level=

If the user does not select this option, it defaults to 1. If the user selects this option with a value higher than 2, they are warned about the risk of increased analysis times.

Second, users can specify thresholds for the number of bytes of shared cache space that counts as potentially problematic cache competition with the following flag:

-t=/–competition-threshold=

If the user does not select this option, it defaults to 256 bytes. If the user selects this option with a value lower than 256, they are warned about the risk of increased analysis times. The user's chosen competition threshold is also used to increase the efficiency of the overall execution of this tool. For example, any functions that are

smaller than the competition threshold are ignored.

Third, users can specify the length of the list of highest-ranked problematic groups of functions that the tool prints out with the following flag:

-r=/–ranking-length=

If the user does not select this option, it defaults to 10.

4.6 Other features

There are numerous features present in this tool that aid usability and transparency. As mentioned above, the user is presented with warnings whenever there is a risk that the options they have chosen will lead to increased analysis times. The user is also presented with warnings when defaults are chosen (e.g., if system variables are missing and they have chosen to suppress empirical cache tests).

Additionally, there is a -h/–help flag that the user can specify which prints information about how to use the tool, the kinds of inefficiencies the tool can identify, and the kinds of configuration options the user can specify. It also goes without saying that there are copious robustness checks in the code that ensure that in the case of unexpected behaviour the programme exits gracefully with an error message rather than crashing. Relatedly, the -k/–keep-temporary-files flag can be useful for diagnosing errors.

Chapter 5

Evaluation

In this section I will evaluate the software contribution of this project illustrated in Chapter 4 in the context of the research contribution illustrated in Chapter 3. First I will evaluate the overall approach I have chosen, namely static analysis and presentation of inefficiencies to the user, in Section 5.1. Next, I will evaluate the functionality of the software itself in light of the kinds of inefficiencies identified in the research contribution: whether it can correctly analyse caches (Section 5.2), and whether it can successfully identify data inefficiencies (Section 5.3) and instruction inefficiencies (Section 5.4). I will also evaluate the robustness and usability features in Section 5.5.

5.1 Overall approach

As discussed briefly in Section 2.4, the overall approach I have taken is to develop a tool which is able to statically analyse source-code files and compiled binary files in order to identify data- and instruction-cache inefficiencies, and to present these analyses transparently to the user alongside suggestions for cache-efficiency optimisations. This is in contrast to other software that currently exists which instead performs cache-efficiency optimisations 'in the background' without presenting its analyses or any alternative approaches to the user.

The major advantage of this approach is that the range of optimisations that can be performed and the care that can be taken regarding optimisations which might counteract each other's effectiveness is significantly improved. Programmers are able to see transparently the cache-efficiency impact of restructuring their code, and the relevant measures of efficiency are presented in a straightforward, easily comparable numerical manner (i.e., it is easy to understand that 6 functions competing for cache space is preferable to 10 functions competing for cache space).

However, there are some drawbacks to this approach. Primarily, the countervailing effect of greater transparency and control is that programmers are required to do more of the legwork in implementing optimisations. While optimising compilers and run-time optimising data-structures allow programmers to make latency improvements without too much effort, the downside of being able to make more sophisticated improvements to code-structure is that it is somewhat more involved

to do so. However, for the advanced programmer of ultra-low-latency software such as HFT systems, latency is the goal and reducing it is the job, so this kind of control and transparency ought to be preferable!

5.2 Cache analysis

The software presents its ordinary cache analysis to the user as follows:

```
==========================================

CACHE ANALYSIS

==========================================


=== L1 data cache ===

Size: 32768 bytes
Linesize: 64 bytes
Associativity: 8
Critical stride: 4096 bytes

=== L1 instruction cache ===

Size: 32768 bytes
Linesize: 64 bytes
Associativity: 8
Critical stride: 4096 bytes

=== L2 cache ===

Size: 262144 bytes
Linesize: 64 bytes
Associativity: 4
Critical stride: 65536 bytes

=== L3 cache ===

Size: 12582912 bytes
Linesize: 64 bytes
Associativity: 16
Critical stride: 786432 bytes

=== L4 cache ===
```

This processor does not have an L4 cache

An important test is whether the empirical cache tests are able to identify correctly the associativities and critical strides. To make sure this functionality worked correctly, I ran the software on a processor with all its dimensions known, but forced it to run the empirical tests by replacing the sysconf() call for instruction-cache associativity in the code with a hard coded -1. With this in place, the following was printed:

=== L1 instruction cache ===

Size: 32768 bytes
Linesize: 64 bytes
This processor does not specify the associativity of its L1 instruction cache.

When I did not choose to suppress empirical tests, the following was printed while the empirical tests were run:

The associativity and critical stride of the L1 instruction-cache are both unknown.
They will now be tested empirically. This may take a couple of minutes.
Assessing instruction-cache associativity empirically.....
Suggested L1 instruction-cache associativity: 8
Assessing instruction-cache critical stride empirically.....
Suggested L1 instruction-cache critical stride: 4096

Setting L1 instruction-cache associativity to 8

Setting L1 instruction-cache critical stride to 4096

This shows that the empirical tests are able to correctly identify the associativity and critical stride of this processor, since these are the known correct numbers. I also ran this on numerous processors, some of which had different dimensions, and these were also identified correctly. For example, my personal laptop's processor has an associativity of 12, which was correctly identified by the empirical tests.

5.3 Data analysis

In order to evalaute the data analysis functionality of the software, I introduced inefficient data layouts to see if they would be noticed. First, I introduced the inefficiently ordered struct from Section 3.4 above, and the software presented me with the following:

=== An inefficient data member ordering has been detected ===

Typename: Inefficient_struct

Current ordering:

char_element;
double_element;
short_element;
int_element;

Current size: 24

Proposed ordering:

double_element;
int_element;
short_element;
char_element;

New size: 16

This ordering saves 8 bytes.

In fact, while developing this tool, I wrote the following class (I have removed function declarations for concision), adding data members in in the order I came across a need for them rather than adding them in an efficient order:

```
struct UDType
{
    std::string name;
    std::string class_info;
    std::vector<variable_info> types_list;
    bool has_auto = false;
    bool has_virtual = false;
    size_t total_size;
    bool is_child = false;
    std::string parent_name = "";
    UDType* parent_class = nullptr;
};
```

The software caught this and gave me the following output:

=== An inefficient data member ordering has been detected ===

Typename: UDType_old

Current ordering:

name;
class_info;
types_list;
has_auto;
has_virtual;
total_size;
is_child;
parent_name;
parent_class;

Current size: 152

Proposed ordering:

name;
class_info;
parent_name;
types_list;
total_size;
parent_class;
has_auto;
has_virtual;
is_child;

New size: 144

This ordering saves 8 bytes.

I also introduced the following inefficiently sized type:

```
struct Inefficient_size
{
    char a[3072];
};
```

The software caught this, and flagged it up in the following manner:

=== An inefficient class size has been detected ===

Typename: Inefficient_size

This type has a total size of 3072 bytes, whose Least Common Multiple with your processor's L1 data-cache critical stride of 4096 bytes is 12288. This means that data members 4 objects apart in contiguous memory will compete with each other for cache space.

Clearly, the programme is able to identify inefficiently sized types and inefficient

orderings (including ones that I didn't notice!) and to suggest appropriate optimisations.

In the case that any inefficiencies are detected, the programme provides the following suggestions, aiding the programmer's task of mitigating the inefficiencies:

> ===== Suggestions =====
>
> It is recommended, unless there is a specific reason to retain an inefficient ordering, that any reorderings identified above are implemented.
> If the overall size of a type may cause problems in the context of access patterns and the critical stride of the data cache, attempting to make the data structure smaller (perhaps by using different data types as sub-members or by using highly compact types like bitfields) is the ideal solution.
> Another mitigation strategy to consider is to avoid contiguous allocations of entire objects when individual data members are accessed in sequence across many of those objects. Instead of these 'arrays of structs', it is sometimes better to organise this data in 'structs of arrays', that is, structs which point to arrays which contain only those data members that are accessed in strided access patterns. This compacts the consecutively accessed data, significantly reducing the risk of cache conflicts.
> Failing that, it may (counterintuitively) be worth trying to make the object slightly larger to reduce the chance of desired accesses evicting useful data from the cache by increasing the Least Common Multiple of the object size and the critical stride.

In this way, the software can identify the kinds of data-cache competition that led to the latency problems identified in Chapter 3 of this report.

5.4 Code analysis

In order to evaluate the code-analysis functionality of the software, I first looked through the code of the software itself (using nm) looking for functions larger than the critical stride of the processor. I then ran the software with its own source code and binary as input, and confirmed that the large functions were correctly identified and flagged up to the user:

> === Large function detected ===
>
> Function name: UserOptions::parse_flags(int, char**)
>
> This function spans between 1 and 2 critical strides. This means that, when it executes, it will occupy 2 cache ways, and may therefore compete with itself for cache space.

=== Large function detected ===

Function name: UserOptions::run_cache_setup()

This function spans between 1 and 2 critical strides. This means that, when it executes, it will occupy 2 cache ways, and may therefore compete with itself for cache space.

=== Large function detected ===

Function name: FileCollection::detect_types(std::bitset⟨8ul⟩&)

This function spans between 1 and 2 critical strides. This means that, when it executes, it will occupy 2 cache ways, and may therefore compete with itself for cache space.

In order to evaluate the software's ability to identify problematic groups of coexecuting functions that compete for cache space, I again looked to Chapter 3 to see if the software could identify known problematic groups. For example, the following is printed when the tool is pointed to a binary file with 32 functions aligned on multiples of 256 bytes (and the ranking length is limited to 2):

===== Function group analysis =====

The following groups of functions have been identified as potential sources of latency problems.
They have been ranked based on a combination of the number of functions in the group, the amount of cache space they compete for, and the number of times they call each other.
This has been calculated based on the instruction-cache's critical stride of 4096 bytes and associativity of 8, as well as a coexecution indirection level of 1 and competition overlap threshold of 1 bytes.

=== Group ===

function_1()
function_17()
main

These 3 coexecuting functions call each other (directly) 2 times and compete for the same 46-byte region of the cache

=== Group ===

function_17()
main

These 2 coexecuting functions call each other (directly) 1 times and compete for the same 46-byte region of the cache

===== Description =====

The largest identified group of coexecuting functions that compete for cache space contains 3 functions.

The size of this group of functions is well below the associativity of the L1 instruction cache, and should therefore not be a source of serious latency issues. It may still be worth trying to reduce the competition between these functions by following the suggestions below.

The following is printed when the tool is pointed to a binary file with 32 functions aligned on multiples of 512 bytes (ranking length still limited to 2):

===== Function group analysis =====

The following groups of functions have been identified as potential sources of latency problems.
They have been ranked based on a combination of the number of functions in the group, the amount of cache space they compete for, and the number of times they call each other.
This has been calculated based on the instruction-cache's critical stride of 4096 bytes and associativity of 8, as well as a coexecution indirection level of 1 and competition overlap threshold of 1 bytes.

=== Group ===

function_1()
function_9()
function_17()
function_25()
main

These 5 coexecuting functions call each other (directly) 4 times and compete for the same 46-byte region of the cache

=== Group ===

function_9()
function_17()

function_25()
main

These 4 coexecuting functions call each other (directly) 3 times and compete for the same 46-byte region of the cache

===== Description =====

The largest identified group of coexecuting functions that compete for cache space contains 5 functions.

The number of groups whose size is close to or equal to the associativity of the L1 instruction cache is: 1.
Such groups of functions may cause an increase in the standard deviation (jitter) among run-time latencies, because there is a risk of cache evictions.

And the following is printed when the tool is pointed to a binary file with 32 functions aligned on multiples of 1024 bytes:

===== Function group analysis =====

The following groups of functions have been identified as potential sources of latency problems.
They have been ranked based on a combination of the number of functions in the group, the amount of cache space they compete for, and the number of times they call each other.
This has been calculated based on the instruction-cache's critical stride of 4096 bytes and associativity of 8, as well as a coexecution indirection level of 1 and competition overlap threshold of 1 bytes.

=== Group ===

function_1()
function_5()
function_9()
function_13()
function_17()
function_21()
function_25()
function_29()
main

These 9 coexecuting functions call each other (directly) 8 times and compete for the same 46-byte region of the cache

=== Group ===

function_5()
function_9()
function_13()
function_17()
function_21()
function_25()
function_29()
main

These 8 coexecuting functions call each other (directly) 7 times and compete for the same 46-byte region of the cache

===== Description =====

The largest identified group of coexecuting functions that compete for cache space contains 9 functions.

The number of groups whose size is close to or equal to the associativity of the L1 instruction cache is: 534.
Such groups of functions may cause an increase in the standard deviation (jitter) among run-time latencies, because there is a risk of cache evictions.

The number of groups whose size is above the associativity of the L1 instruction cache is: 1.
Such groups of functions are likely to cause an increase in latencies, because they will evict each other from the cache.

Clearly, the software can correctly identify the kinds of instruction-cache competition that led to the latency problems identified in Chapter 3. As a more realistic example, the following groups were identified when the software was run on its own source code and compiled binary (ranking length limited to 2 again):

===== Function group analysis =====

The following groups of functions have been identified as potential sources of latency problems.
They have been ranked based on a combination of the number of functions in the group, the amount of cache space they compete for, and the number of times they call each other.
This has been calculated based on the instruction-cache's critical stride of 4096 bytes and associativity of 8, as well as a coexecution indirection level of 1 and competition overlap threshold of 256 bytes.

=== Group ===

main
UserOptions::parse_flags(int, char**)
UserOptions::run_cache_setup()
UserOptions::run_analysis()

These 4 coexecuting functions call each other (directly) 3 times and compete for the same 433-byte region of the cache

=== Group ===

FileCollection::get_alignment(std::__cxx11::basic_string⟨char, std::char_traits⟨char⟩, std::allocator⟨char⟩ ⟩)
FileCollection::detect_types(std::bitset⟨8ul⟩&)

These 2 coexecuting functions call each other (directly) 2 times and compete for the same 1984-byte region of the cache

In the case that any inefficiencies are detected, the tool describes its analysis and provides suggestions as follows (this particular output is from running the tool on the latest version of its own code):

===== Description =====

The largest identified group of coexecuting functions that compete for cache space contains 4 functions.

The size of this group of functions is well below the associativity of the L1 instruction cache, and should therefore not be a source of serious latency issues. It may still be worth trying to reduce the competition between these functions by following the suggestions below.

===== Suggestions =====

There are a number of things that programmers can do to mitigate problems that might be identified in this analysis.

(1) rewrite the functions to involve less code!
(2) inline small functions that may evict parts of larger functions from the cache so that they do not conflict for cache space
(3) combine smaller functions that compete for cache space into fewer, longer functions
(4) move functions around in the code so that functions that are called

> together are placed together in memory and will therefore occupy non-overlapping regions of the cache
> (5) try the gcc flag –falign-functions=⟨alignment⟩, which will force functions onto the specified alignment and which may improve the situation - but may also make it worse!

5.5 Robustness and usability

The software contains many checks to ensure that unexpected input does not crash the programme or cause any strange behaviour. For example, the following is printed if no files are specified on the command line and the -c/–cache-info-only option is not selected:

> No files specified!
> Normal usage: ./latency_tool [–options ...] [source_code_files ...] binary_file
> For cache info only, invoke with the -c/–cache-info-only option.

If a file is specified that the programme cannot find, the following is printed before any processing or analysis is started:

> The file ‘does_not_exist.txt’ does not exist

There are also other features that aid usability. For example, the programme warns users whenever defaults are chosen. The following is printed if a system variable is missing but the user has chosen to suppress empirical tests:

> The following cache dimensions are unknown, but you have chosen not to execute empirical tests:
> – L1 instruction cache associativity
> Associativities will default to 8, linesizes to 64, and critical strides to 4096. This can produce results that are inaccurate for your processor, but the suggestions may still prove useful.

Warnings about long executions times are are also printed when the user selects a competition threshold below 256 bytes or a coexecution indirection level of more than 2. The software also contains a -h/–help flag, which describes the functionality of the tool, instructs the user about ordinary usage, and enumerates all possible user-configuration options.

The software has some shortcomings in its current form. For example, it is somewhat tedious to specify all the source code files of very large projects. It would be desirable to implement functionality whereby the user can perhaps specify a single file that contains a list of source code file names, or can specify a folder in which to analyse all available files (or perhaps files with appropriate extensions such as .c, .cpp, .h, and .hpp), or can specify a build automation file (such as a Makefile) that is used to build the project and that the software can analyse to discover relevant

filenames. Another shortcoming is that very long execution times are possible. This is a result of the way in which the software currently identifies groups of functions, and I am certain that it would be possible to optimise the current algorithm. As it stands, however, I have found that a full analysis of most ordinary-sized programmes is perfectly achievable within a minute or two.

Chapter 6

Conclusions and future directions

This report has shown that cache-efficiency is of great importance to programmers of low-latency software such as High Frequency Trading systems. The structure of the processor cache, the data structures used, and the layouts of data and instructions in memory all contribute to the cache efficiency of a piece of software. Many of these considerations are affected by the structure of C++ code but are opaque to C++ programmers. This report has also covered the development of a piece of software that makes these opaque inefficiencies transparent to programmers in order that they can target them in the interests of achieving low latencies and low standard deviations among latencies. This software contribution has been shown to correctly identify the inefficiencies that were identified in the research contribution.

Each of these contributions suggest many potentially fruitful avenues for future research. While Section 3.2 focusses on L1 data- and instruction-cache associativities and critical strides, linesizes and the structure of lower level caches are factors that ought to be investigated in more detail. In the context of High Frequency Trading, it could be argued that if you are dealing with lower level cache misses, something has already gone wrong; nevertheless, in the context of low-latency software in general, lower level cache efficiency is a significant factor affecting execution latencies.

Section 3.3 covered some C++ containers and their relative cache efficiencies. However, the range of available types of container is large and there are many STL alternatives beyond those that I discussed. In order to submit my conclusions to more rigorous empirical tests, it would be worth investigating whether the latency-rankings indicated in broad comparisons (e.g., [50]) can be explained by the factors I identify as dominant — namely the number of instructions requiring execution and number of required references to the data-cache — and whether the patterns found in cache miss percentages and instructions per cycle, for example, hold across other contiguous and discontiguous container implementations. This would involve similar tests to those carried out in Section 3.3 across larger numbers of data-structure alternatives providing the same general functionality. With respect to the software contribution, since the (dis)contiguity of data structures has been seen to be a significant factor affecting latency patterns, the efficacy of the software contribution could be augmented by functionality which is able to identify discontiguous data structures and suggest contiguous alternatives to use in their place.

The results found in Section 3.4 are interesting and complex and likewise sug-

gest many research directions that could be taken. One obvious one based on the conclusions I have drawn and discussed would be to investigate in more detail the pattern observed in vectors and arrays that hold packed and unpacked elements. If my conclusion that misaligned accesses incur a latency penalty is correct, are unaligned reads worse than aligned writes? If so, why? If my conclusion is wrong, what is the real reason that 16-byte packed and unpacked structs in arrays behave differently, and why does this difference disappear when optimised? In order to answer these questions, I would suggest testing packed and unpacked structs outside the context of sequence containers, e.g., in associative containers such as sets and maps. Correspondingly, it would be interesting to extend the empirical-cache analysis idea already implemented in the software as it currently is to testing empirically whether misaligned accesses incur a penalty on the user's specific processor. If [48] is correct and there exist processors which have eliminated misaligned access penalties, this further empirical test could be used to suggest that, where appropriate, the user instructs their compiler to pack their structs.

In sum, as a consequence of the dearth of existing literature on cache-efficiency for ultra-low-latency software and of existing tools that can help programmers to identify and mitigate such inefficiencies, there are many avenues for future empirical research that would not only be interesting in their own right, but would also facilitate the extension and improvement of software of the kind that I have developed.

Bibliography

[1] Aldridge I. High-frequency trading : a practical guide to algorithmic strategies and trading systems. Wiley; 2013. pages 2, 5, 6

[2] Cartea A, Jaimungal S, Penalva J. Algorithmic and high-frequency trading. Cambridge University Press; 2015. pages 2

[3] Goldstein K. Building Low Latency Trading Systems; 2018. Available from: https://www.youtube.com/watch?v=yBNpSqOOoRk. pages 2, 5

[4] Cook C. CppCon 2017: Carl Cook "When a Microsecond Is an Eternity: High Performance Trading Systems in C++"; 2018. Available from: https://www.youtube.com/watch?v=NH1Tta7purM. pages 2, 5, 8

[5] Pusz M. code::dive 2017 – Mateusz Pusz – Striving for ultimate low latency; 2018. Available from: https://www.youtube.com/watch?v=vzDl0Q91MrM. pages 2, 6, 43, 56

[6] Carruth C. CppCon 2014: Chandler Carruth "Efficiency with Algorithms, Performance with Data Structures"; 2014. Available from: https://www.youtube.com/watch?v=fHNmRkzxHWs. pages 2, 29

[7] Sapir N. Core C++ 2019 :: Nimrod Sapir :: High Frequency Trading and Ultra Low Latency development techniques; 2019. Available from: https://www.youtube.com/watch?v=_0aU8S-hFQI. pages 2

[8] Stroustrup B. An overview of C++. In: Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming; 1986. p. 7-18. pages 3, 6

[9] Stroustrup B. The design and evolution of C++. Pearson Education India; 1994. pages 3, 6

[10] Stepanov A, Lee M. The standard template library. vol. 1501. Hewlett Packard Laboratories 1501 Page Mill Road, Palo Alto, CA 94304; 1995. pages 3, 7

[11] Plauger PJ, Lee M, Musser D, Stepanov AA. C++ standard template library. Prentice Hall PTR; 2000. pages 3, 7

[12] Hill MD, Smith AJ. Evaluating associativity in CPU caches. IEEE Transactions on Computers. 1989;38(12):1612-30. pages 3, 9

[13] Wu Y. Ordering functions for improving memory reference locality in a shared memory multiprocessor system. ACM SIGMICRO Newsletter. 1992;23(1-2):218-21. pages 3, 10, 59

[14] Calder B, Krintz C, John S, Austin T. Cache-conscious data placement. In: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems; 1998. p. 139-49. pages 3, 10, 11

[15] Hwu WmW, Chang PP. Achieving high instruction cache performance with an optimizing compiler. In: Proceedings of the 16th Annual International Symposium on Computer Architecture; 1989. p. 242-51. pages 3, 11, 59

[16] Chilimbi TM, Hill MD, Larus JR. Cache-conscious structure layout. In: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation; 1999. p. 1-12. pages 3, 10, 11

[17] Ostrovsky I. Gallery of Processor Cache Effects; 2010. Available from: `https://igoro.com/archive/gallery-of-processor-cache-effects/`. pages 3, 10

[18] Sandler A. Aligned vs. unaligned memory access - Alex on Linux; 2008. Available from: `http://www.alexonlinux.com/aligned-vs-unaligned-memory-access`. pages 3, 10, 29, 43

[19] Ahn SH. Data Alignment; 2022. Available from: `https://www.songho.ca/misc/alignment/dataalign.html`. pages 3, 10

[20] Lemire D. Data alignment for speed: myth or reality?; 2012. Available from: `https://lemire.me/blog/2012/05/31/data-alignment-for-speed-myth-or-reality/`. pages 3, 10, 29, 43

[21] attractivechaos. Does packed struct hurt performance on x86_64?; 2013. Available from: `https://attractivechaos.wordpress.com/2013/05/02/does-packed-struct-hurt-performance-on-x86_64/`. pages 3, 10, 29, 43

[22] GDB: The GNU Project Debugger; 2022. Available from: `https://sourceware.org/gdb/`. pages 3, 10, 46

[23] objdump(1) - Linux manual page; 2022. Available from: `https://man7.org/linux/man-pages/man1/objdump.1.html`. pages 3, 10, 46

[24] binutils/objdump.c; 2022. Available from: `https://github.com/CyberGrandChallenge/binutils/blob/master/binutils/objdump.c`. pages 3, 10, 46

[25] nm(1): symbols from object files - Linux man page; 2022. Available from: `https://linux.die.net/man/1/nm`. pages 3, 10, 46

[26] binutils/nm.c; 2022. Available from: `https://github.com/CyberGrandChallenge/binutils/blob/master/binutils/nm.c`. pages 3, 10, 46

[27] Perf Wiki; 2022. Available from: https://perf.wiki.kernel.org/index.php/Main_Page. pages 3, 10, 12, 46

[28] Valgrind; 2022. Available from: https://valgrind.org/. pages 3, 12

[29] Cachegrind; 2022. Available from: https://valgrind.org/docs/manual/cg-manual.html. pages 3, 11

[30] Jones CM. What do we know about high-frequency trading? Columbia Business School Research Paper. 2013;(13-11). pages 5

[31] Menkveld AJ. The Economics of High-Frequency Trading. Annual Review of Financial Economics. 2016;8:1-24. pages 5

[32] Cartea Á, Penalva J. Where is the value in high frequency trading? The Quarterly Journal of Finance. 2012;2(03):1250014. pages 5

[33] Gomber P, Haferkorn M. High frequency trading. In: Encyclopedia of Information Science and Technology, Third Edition. IGI Global; 2015. p. 1-9. pages 5

[34] Canning P, Cook W, Hill W, Olthoff W, Mitchell JC. F-bounded polymorphism for object-oriented programming. In: Proceedings of the fourth international conference on functional programming languages and computer architecture; 1989. p. 273-80. pages 6

[35] Coplien JO. Curiously recurring template patterns. In: C++ gems; 1996. p. 135-44. pages 6

[36] Veldhuizen TL. C++ templates are turing complete. Citeseer; 2003. pages 6

[37] Bentley M. Introduction of std::hive to the standard library; 2021. Available from: https://isocpp.org/files/papers/P0447R16.html. pages 8

[38] Bentley M. mattreecebentley/plf_hive; 2022. Available from: https://github.com/mattreecebentley/plf_hive. pages 8

[39] Bentley M. PLF Library - colony; 2021. Available from: https://plflib.org/colony.htm. pages 8

[40] Hopscotch hashing; 2022. Available from: https://tessil.github.io/2016/08/29/hopscotch-hashing.html. pages 8

[41] Goetghebuer-Planchon T. Tessil/hopscotch-map; 2022. Available from: https://github.com/Tessil/hopscotch-map. pages 8

[42] Herlihy M, Shavit N, Tzafrir M. Hopscotch Hashing. Lecture Notes in Computer Science. 2008;5218:350-64. pages 8

[43] M LA. robin_hood unordered map & set; 2022. Available from: https://github.com/martinus/robin-hood-hashing. pages 8

[44] Celis P. Robin hood hashing. University of Waterloo; 1986. pages 8

[45] Celis P, Larson PA, Munro JI. Robin hood hashing. In: 26th Annual Symposium on Foundations of Computer Science (sfcs 1985). IEEE; 1985. p. 281-8. pages 8

[46] Norvig P. Teach Yourself Programming in Ten Years; 2022. Available from: https://norvig.com/21-days.html#answers. pages 8

[47] U D. Memory part 2: CPU caches; 2007. Available from: https://lwn.net/Articles/252125/. pages 8, 9

[48] Fog A. The microarchitecture of Intel, AMD and VIA CPUs An optimization guide for assembly programmers and compiler makers; 2022. Available from: https://www.agner.org/optimize/microarchitecture.pdf. pages 9, 43, 75

[49] Pettis K, Hansen RC. Profile guided code positioning. In: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation; 1990. p. 16-27. pages 10, 59

[50] Ankerl M. Hashmaps Benchmarks - Overview; 2022. Available from: https://martin.ankerl.com/2019/04/01/hashmap-benchmarks-01-overview/. pages 10, 74

[51] katecpp. Struct members order does make a difference; 2015. Available from: https://katecpp.github.io/struct-members-order/. pages 10, 29

[52] Wagner L. Saving a Third of Our Memory by Re-ordering Go Struct Fields; 2020. Available from: https://wagslane.dev/posts/go-struct-ordering/. pages 10, 29

[53] Benchmark; 2022. Available from: https://github.com/google/benchmark. pages 12

[54] ARM: Unaligned Access Gives Unexpected Results;. Available from: https://developer.arm.com/documentation/ka003038/latest. pages 30

[55] Rupp K. Strided Memory Access on CPUs, GPUs, and MIC — Karl Rupp; 2022. Available from: https://www.karlrupp.net/2016/02/strided-memory-access-on-cpus-gpus-and-mic/. pages 43, 56

[56] GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF); 2019. Available from: https://gcc.gnu.org/. pages 46

[57] gcc-mirror/gcc; 2022. Available from: https://github.com/gcc-mirror/gcc. pages 46

[58] Chomsky N. Three models for the description of language. IRE Transactions on information theory. 1956;2(3):113-24. pages 52

[59] llvm-dwarfdump - dump and verify DWARF debug information — LLVM 16.0.0git documentation; 2019. Available from: `https://llvm.org/docs/CommandGuide/llvm-dwarfdump.html`. pages 53

[60] Lam MD, Rothberg EE, Wolf ME. The cache performance and optimizations of blocked algorithms. ACM SIGOPS Operating Systems Review. 1991;25(Special Issue):63-74. pages 56