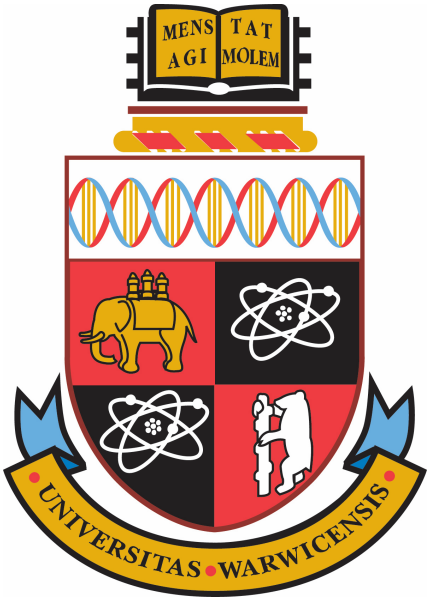
Detection and Optimisation of Data Structures in Compile Time

CS351 Final Report

Mahiethan Nitharsan



University of Warwick

Department of Computer Science

Supervised by Dr Richard Kirk

April 2024

Abstract

As processor speeds improved over the past few decades, memory performance has not kept up in pace, resulting in a large performance gap. This ultimately affects memory-bound programs with frequent memory reads and writes, such as physics simulations. The aim of this project is to create compiler passes that are able to optimise data structures, with the aim to improve the memory-bound performance of programs. This is achieved by detecting specific data structures within the program, then applying valid optimisations to them in order to reduce execution times and memory consumption. The challenge comes with detecting data structures that could be declared in numerous different ways and also finding out which optimisations are compatible with each other, to ensure that the intended functionality of the source program is not broken. After applying different optimisations, execution times have been reduced by around 10 - 77% and memory reductions at around 1 - 46% are also seen. Even though the developed compiler passes focus on detecting and optimising simple data structures, this project lays the groundwork for future extensions that would encompass more complex data structures and optimisations.

Keywords

Memory-bound performance, Optimisation, Data structures, Compiler pass, LLVM compiler, C++

Contents

1 Introduction

1.1 Motivation

In 1965, Gordon Moore proposed the *Moore's Law*, stating that the number of transistors in integrated circuits will double every two years [1, p. 5]. This concept has set expectations of exponential growth in processor performance over time. These expectations have been fulfilled as significant enhancements in processor performance were seen between 1986 and 2005, as depicted in Figure 1. The performance line for the processor measures the average increase in memory requests per second (on a logarithmic scale) [1, p. 78]. Higher values on the processor performance scale correlate with lower processor latency, indicating that greater number of operations can be executed per unit time.

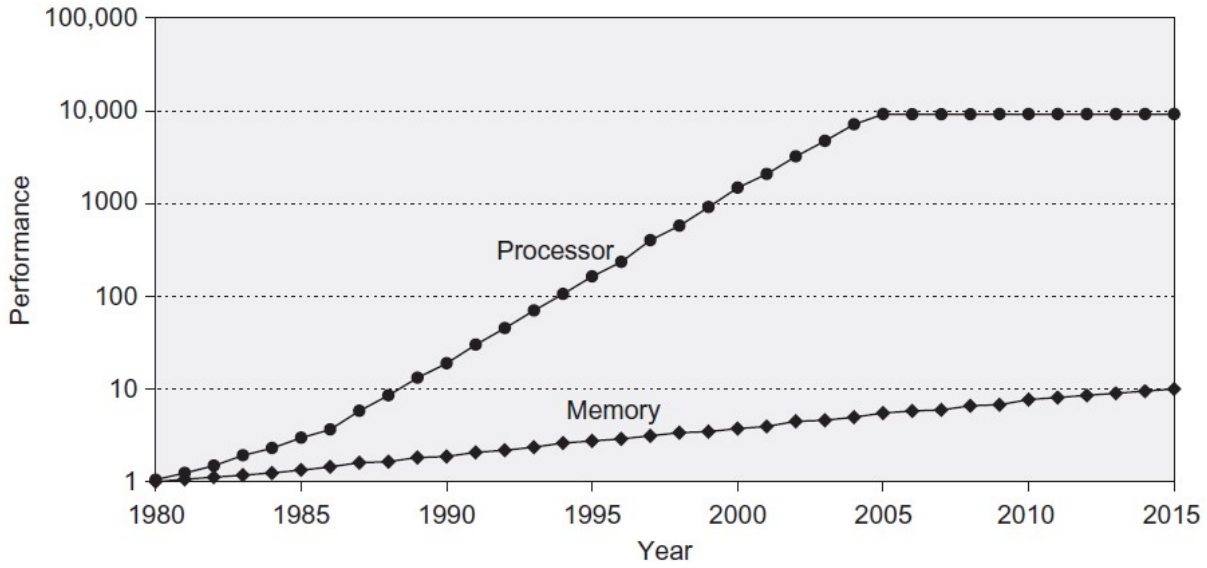


Figure 1: Performance comparison between processors and main memory, as the number of transistors in integrated circuits increase over the years. [1, p. 80]

Over these years, the number of transistors within a processor is increased by reducing the actual sizes of the transistor technology used (referred to as process node). Not only does this increase the number of transistors per unit area within the CPU, this enables the CPU to run at a faster clock speed and also reduces the delays between the logic gates [2]. These two benefits increase the number of CPU operations per unit time and reduce the time taken to execute tasks respectively, which explains the increase in number of memory accesses per unit time in Figure 1.

Looking back at Figure 1, the performance line for main memory shows the increase in DRAM accesses

per second (on a logarithmic scale), which inversely correlated with DRAM access latency [1, p. 78]. The higher the value for the memory performance, the lower value for memory access latency. Unlike processor performance, memory performance has not increased at the same rate. The increased number of transistors within memory does not produce large performance benefits as seen with processors.

With processor performance constant at around 10,000 in the year 2015, it is still 1000x greater than main memory, producing a large performance gap. This performance gap will likely to be present in the year 2024 given the slow rate of improvement, meaning that current overall system performance is bottle-necked by weak memory performance, in particular, slow memory access times.

Hardware and software optimisations exists that attempt to bridge this performance gap. Hardware optimisations are the most effective at reducing memory latency/access times. Most of these optimisations target the cache memory, which uses faster SRAM technology compared to DRAM technology used in main memory. Methods such as increased cache sizes, pipelined caches and multi-level caches aim to reduce the average memory access times, but comes with heavy costs.

SRAM memory costs roughly ten times more to produce per bit than DRAM, so it would lead to higher monetary costs to employ [1, G-25]. Additionally, power consumption may be considered, especially for systems with constrained power budgets where cache usage can take up to 25% to 50% of total power consumption [1, p. 80]. Therefore, hardware designers may find it difficult to apply such hardware optimisations since they need to find a right balance between performance, monetary costs, and power consumption.

Software optimisations, on other hand, focus on achieving the best utilisation of the current hardware resources, in order to reduce memory access times. Compared to hardware optimisations, these methods do not require any additional hardware so it does not have any monetary costs and will have a smaller impact on power consumption. The focus is on modifying the source code of programs where simple optimisations are often applied by the programmer. However, most meaningful optimisations are typically left out by the programmer since they are more difficult to implement by hand. Instead, programmers focus on implementing the functionality of the code and often rely on the compiler to apply the necessary software optimisations.

Compilers such as GCC [3] and LLVM [4] contain several optimisations/compiler passes that can be ap-

plied to the source code respectively. Most of these focus on increasing the *compute bound performance* of the program. Examples include loop based optimisations, dead code elimination and parallelisation; these all focus on increasing the number of CPU operations per unit time. Although boosting CPU performance is beneficial, the main target is to optimise memory performance, so it can perform as well as the CPU. Unfortunately, there is a lack of available *memory-bound optimisations*, which aim to reduce the memory access times. Specifically, optimisations that target data structures are not present at all. Data structures within programs may store large amounts of data and may be involved in frequent memory read and write operations, so optimising these to access memory faster will indeed boost memory-bound performance.

A reason why memory optimisations may not exist for data structures is because they may be implemented in a specific way by the programmer, to ensure that the program functions in a way they intended. Even though these data structures can be optimised in terms of memory layout and consumption, altering these can affect the program's functionality after compilation of the code. This is undesirable in programs that are expect to run for continuous periods of time and require precise results to be produced, such as physics simulations. Despite this, the option to apply memory optimisations, especially for data structures, should still exist since it is ultimately up to the programmer's decision to use these optimisations on their code.

1.2 Project Goals

Given this problem statement and the existing performance gap between processors and main memory [1, p. 80], the aim of this project is to improve memory-bound performance of C programs. This will be done by implementing optimisations that are applied to detected data structures.

Detection passes will be created to identify certain types of data structures within a program. Once detected, optimisation passes will be developed and applied onto these data structures. To ensure that the program's functionality is not affected in any way, the optimisations must be checked to see if they can safely be applied to the data structures within the program.

The LLVM Compiler Infrastructure will be used to create the compiler passes, that are able to detect and optimise data structures. More details on LLVM and compiler passes will be further discussed in detail in Section 2.

1.2.1 Objectives

The project objectives defined in the specification report was adapted as the project progressed. Each objective was given priorities (Must, Should, Could, Won't) based on the MoSCoW prioritisation technique [5], to determine parts of the project are the most important and should be completed first in the project timeline.

R1.1 The project **MUST** detect Array of Structures (AoS).

R1.2 The project **MUST** detect Structures of Arrays (SoA).

R1.3 The project **MUST** detect Array of Structure of Arrays (AoSoA).

R1.4 The project **SHOULD** detect Linked List data structures.

R1.5 The project **SHOULD** detect Tree data structures.

R2.1 The project **MUST** optimise Array of Structures (AoS).

R2.2 The project **MUST** optimise Structures of Arrays (SoA).

R2.3 The project **MUST** optimise Array of Structure of Arrays (AoSoA).

R2.4 The project **SHOULD** optimise Linked List data structures.

R2.5 The project **SHOULD** optimise Tree data structures.

R3.1 Each optimisation **MUST** ensure functional correctness.

R3.2 Observations **MUST** measure run-time performance for each optimisation.

R3.3 Observations **MUST** measure memory usage for each optimisation.

R3.4 The project **WON'T** optimise functions that use the detected data structures.

Figure 2: Final list of project objectives

Each of the following objectives in Figure 2 will be evaluated on its completion in Section 6.3.

1.3 Project Challenges

There are several challenges that are involved in the completion of this project. The following challenges are from different aspects of the project and were identified before project initiation. By doing this, it gives a better guide on the required methods and techniques for successful project completion.

1.3.1 Research on LLVM

The first problem with this project is the lack of experience with the LLVM toolchain used to create and use the compiler passes. In particular, time is required to learn the special class hierarchy and functions required for implementation in C++. Dedicating time at the start to research and practice using the tools is not feasible given the restricted project timeline. Without good prior expertise, implementation of the compiler passes can be very difficult and could result in incorrect development, especially with more complex implementations. Aside from the available reference guides, there may also be very little help from forums and blogs that could give handy solutions to issues that may be encountered.

To overcome this issue, the iterative development plan discussed in Section 5 will not require this initial planning and research phase. Instead, as each detection and optimisation pass is being developed, experience and knowledge on the usage the LLVM tools will be gained as the project progresses. With good experience programming in C and C++, it would be quick and easy to remember the useful functions and classes required for this project, as they will be commonly used in the development of the compiler passes.

1.3.2 Functionality of Optimisations

A main challenge within this project is to check whether the implemented optimisations can be safely applied to the data structures. This means testing each optimisation on various different declarations and usages of the data structure. Depending on how much research is available on the optimisation, testing each optimisation on various types of data structures may consume more time, however this is necessary to make sure optimisations are valid for certain declarations.

Not only should each optimisation be checked individually, a combination of optimisations should also be tested together. It may be the case that two or more specific optimisations cannot be applied together without breaking the program's code. Therefore, these scenarios should be detected and necessary checks

should be added within the compiler passes to prevent such optimisations from clashing with each other.

1.3.3 Scope of Detection

In terms of the detection, there are various ways data structures are declared and used within the program. When implementing the detection of one particular data structure, the time and effort required to detect all variants of that data structure may introduce project delays, which interrupts the flow of the project and could lead to the project not being completed in time.

To avoid this, research must be conducted on the number of different data structure declarations. If this number is large, the scope of the detection can be reduced to only detect a few commonly-used examples. This makes sure that some implementation of the detection passes has been completed, without causing unnecessary project delays that will affect the completion of the other key detection and optimisation passes.

2 Background

To execute a program written in a particular language on a target machine, the program needs to be translated to another target language; this translation is done by a *compiler* [6, p. 1]. Compilers are required to translate a program to machine code, which forms an executable that can be run by the user.

Figure 3 shows the structure of a two-phase compiler that is able to translate C programs into machine code. The first phase of this compiler is referred to as the *front-end*, which is responsible for analysing the source language. It performs lexical analysis and semantic analysis on the source language, which ultimately produces an intermediate representation (IR) [7, p. 6].

The front-end can produce several intermediate representations of the source program, that could either be high or low-level representations. High-level IR is close to the source language and an example of this is the abstract syntax tree (AST). The AST is high-level because it represents the hierarchical structure of the source program, which is useful for checking types and making sure the semantics are correct. Low-level IR, such as three-address code, is close to the target machine code and is suitable for machine independent tasks like register allocation and instruction selection [6, p. 358].

The *back-end* phase of the compiler may utilise one or a sequence of the IRs to generate the machine code, by mapping the IR onto the instruction set and finite resources of the target machine [7, p. 7]. This results in an executable that performs the intended functions exactly defined by the source code.

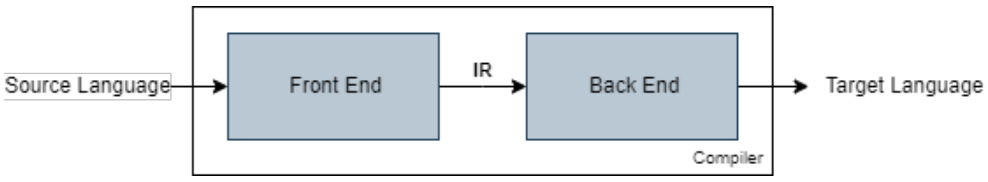


Figure 3: Structure of a two-phase compiler

The presence of IR means that new phases can be added within compilation. One useful phase is the *optimiser*, which is added between the front-end and the back-end to form the three-phase compiler as shown in Figure 4. The optimiser phase is an IR-to-IR transformer that tries to improve the IR program, by making multiple passes over the IR to analyse and/or re-write it. The objectives of the optimiser is to make the back-end produce a faster, smaller and/or power-efficient program [7, p. 8].

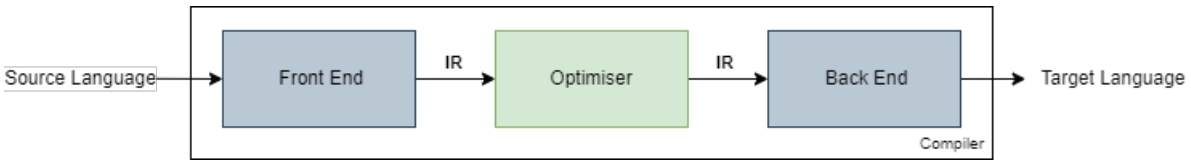


Figure 4: Structure of a three-phase compiler

This particular functionality of the three-phase compiler will be the main focal point in this project, as it will be used to improve the memory layout of data structures in C-language programs, which will ultimately improve memory consumption and execution times. The three-phase compiler that will be used in this project is the LLVM Compiler Infrastructure.

2.1 LLVM

The LLVM Compiler Infrastructure is not a compiler on its own, instead it is a collection of modular and reusable compiler and toolchain technologies [8]. In this collection, the necessary tools are included to produce the same functions as the front-end, back-end and optimiser phases of a typical three-stage compiler. The Clang tool is used as the front-end to parse the source programs written in C, and convert it to an intermediate representation called LLVM IR [9]. Additionally, the LLVM Core libraries is used for the back-end and optimisation tools, to operate on the LLVM IR.

Figure 5 shows the overall structure of the compiler using the LLVM tools. The optimisation phase in LLVM is called the *pass pipeline*, which contains a sequence of one or more *compiler passes* that performs analysis or transformations to the IR. Several analysis and transformation passes are available for use, such as obtaining loop information and dead code elimination respectively [4].

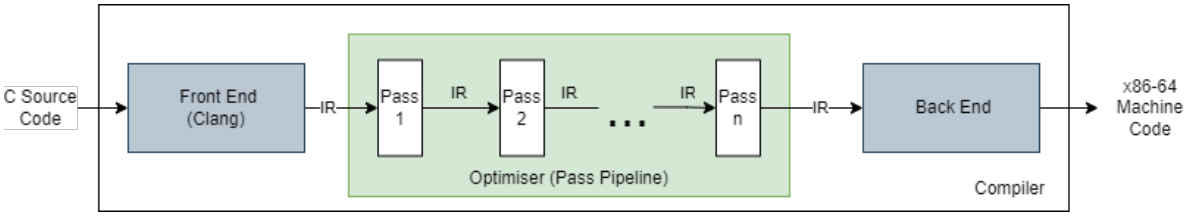


Figure 5: Three-phase compiler created using LLVM

As mentioned previously, there is a lack of memory optimisations available in the LLVM library. Therefore, the project will develop on this library, to implement compiler passes using the LLVM C++ API that apply memory optimisations. The pass pipeline will be predominantly used to apply these new compiler passes, on it own or in sequence with other passes, to the source program.

2.1.1 LLVM IR

The intermediate representation used by the LLVM compiler tools is referred to as LLVM IR. This is a three-address code representation that is human-readable and is very similar to assembly language. There are two forms of the LLVM IR: the assembly language form with a file extension `.ll` and its equivalent LLVM bytecode with the `.bc` file extension. The LLVM bytecode is simply a binary representation of the LLVM IR which is converted into the target language by the back-end of the compiler. The project mainly uses the `.ll` file as the compiler passes will be able to read each instruction of the IR.

A unique feature about LLVM IR, compared to standard assembly language, is how the instructions are organised, as shown in Figure 6. The whole source file is referred to as a Module, which contains all the code. Each Module contains one or several Functions, which are chunks of code representing function bodies. Each Function may contain one or several BasicBlocks, which are contiguous chunks of instructions. Within a Function, several BasicBlocks may be present when a for loop is present and is branching between different BasicBlocks. Lastly, each BasicBlock contains instances of an Instruction, which is a single code of operation, such as load/store operation in memory and arithmetic or casting operations [10].

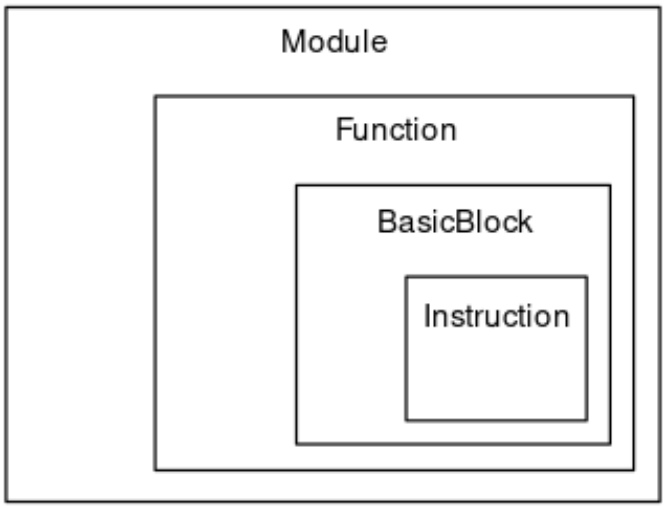


Figure 6: Overall structure of LLVM IR [10]

2.1.2 LLVM C++ API

Several APIs exist that enable programmers to create compiler passes that analyse and/or transform the IR. The LLVM C++ API is the most detailed and commonly used library and is used in this project. It

defines a class hierarchy, which provides a thorough list of classes and interfaces to interact with the IR using C++ [11].

Figure 7 shows the most commonly used super classes in this project and their associations. The `Value` class is the most important in the LLVM source base, as it is the super-class of all LLVM classes associated with functions, instructions and operands. The `User` class is a subclass of `Value` which exposes a list of operands of the current User value. The most important User subclasses used in this project include the `Instruction` class, which defines a wide range of different instruction types such as `LoadInst` and `StoreInst`, and the `Constant` class, which defines `Function` instances and global variables within the program as `GlobalValue` instances. The `BasicBlock` subclass to `Value` is used to create new BasicBlocks of IR instructions, as part of the optimisations [12].

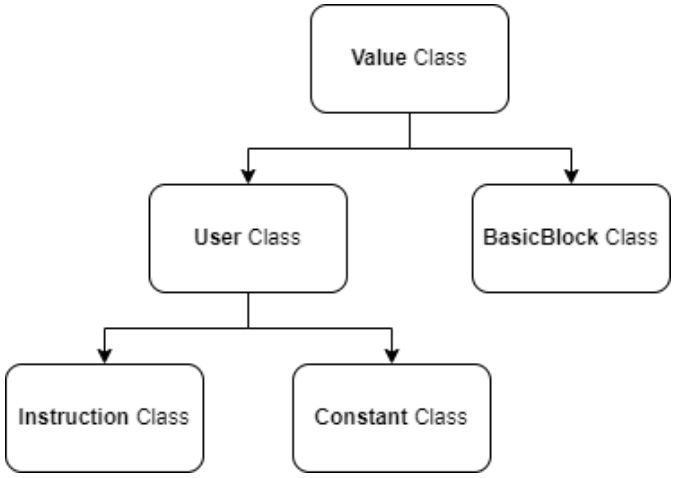


Figure 7: Hierarchy of commonly-used LLVM core classes.

Every `Value` instance has an associated `Type` class. Figure 8 shows the commonly used Type subclasses that are used within the compiler pass, such as `StructType` and `ArrayType`, which is useful for identifying structs and arrays respectively.

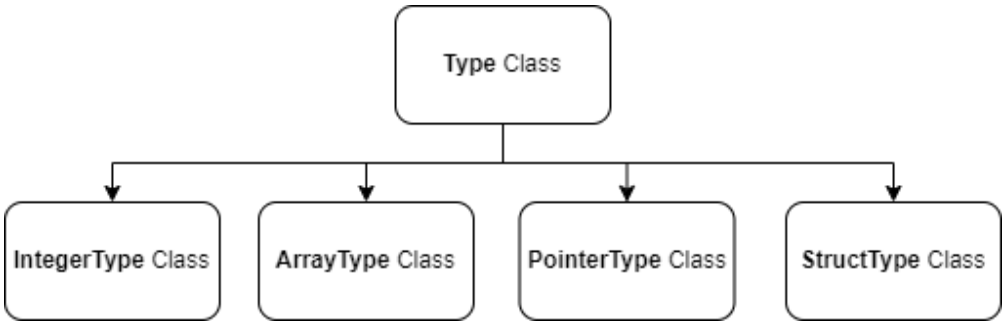


Figure 8: Hierarchy of commonly-used LLVM Type classes.

2.1.3 LLVM / Clang tools

The required LLVM tools are accessed with a wide range of commands using a command line interface. The commands used in this project are responsible for performing the three main operations of the three-stage compiler: convert source code to LLVM IR, apply compiler passes to the IR and generate machine code from the IR.

The Clang front-end tool is used to convert the C source code to its equivalent LLVM IR. This is done using the command `clang`, with the flag `-emit-llvm` [13]. To create the LLVM IR in assembly or bytecode representations, an additional flag `-c` or `-S` is required respectively. Listing 1 shows an example of producing an LLVM IR from a source program.

Listing 1 General format for using the **clang** command to produce LLVM IR as assembly

```
clang -emit-llvm static_AoS.c -c -o staticAoS.ll
```

Once the LLVM IR is created, this is then used by the optimiser, which is the `opt` tool in LLVM. The `opt` tool takes in the compiler pass as a shared object file (`.so` file) using the command line flag `-load-pass-plugin`. One top-level compiler pass (called `allPasses`) will be created and would run the other compiler passes, depending on what pass name was entered in the `-passes` flag. Other optional flags can also be used to provide additional features or information, such as the `-time-passes` flag which prints out the total optimisation time and the order of execution of each compiler pass within the pass pipeline.

Listing 2 is the format of the command used to run the LLVM optimiser tool. The input IR file, represented by the file `fileToTest.bc`, can either be in bytecode (`.bc`) or assembly (`.ll`).

Listing 2 General format for using the **opt** command to run a compiler pass

```
opt -load-pass-plugin=../../location/to/pass -passes="passName" -time-passes <
↪  fileToTest.bc > /dev/null
```

`\dev\null` in Listing 2 referred to the output file that can produced as `.bc` file or, in this case, no output file can be produced. The back-end of the compiler requires a `.bc` file to produce an executable of the optimised program, as shown in Listing 3, which is a binary file that cannot be read. To read the optimised program in its assembly form, the output bytecode can be disassembled into the equivalent `.ll` file using the `llvm-dis` tool [14].

Listing 3 Format of **clang** command to produce executable of optimised program

```
clang optimised.bc -o optimised
```

These are the main three LLVM tools that will be used to run the newly-created compiler passes and test its functionality on C programs.

2.2 Data structures in C

A data structure is a particular arrangement of data in memory. It is classified into two different types: primitive and non-primitive. Basic data types such as integer, floats and characters are primitive data structures, meaning that they store data of one type only. The main focus of this project is on non-primitive data structures, which are able to store multiple data types [15, p. 46].

More specifically, the non-primitive data structures in C, that are detected and optimised in this project, is built up of structure ('struct') elements. This is because the layout of these multiple data types will become more complex in memory, and this is the major focal point in memory layout optimisation for this project.

2.2.1 Structure (Struct)

In C programs, a *structure* is a user-defined data type that contains various fields of data. This term is also abbreviated to 'struct', to avoid confusion with the term 'data structure', and this abbreviation will be used through the report to refer to this structure data type. Listing 4 shows an example of a single struct in C.

Listing 4 Example of a **struct** in C

```
struct elem {
    int a;
    float b;
    struct elem* next;
};
```

Each variable inside a struct is referred to as a field, which can be any data type. This makes it useful when accessing multiple related data values, which require the same operation to be performed on. Since all struct fields are stored contiguously in memory, there is no need to retrieve each data field individually. Additionally, structures with included pointer fields, as seen in Listing 4, allows for the creation of abstract data types (ADTs) such as linked lists and trees.

Despite the fields of a structure being stored contiguously in memory, various optimisations can be made to each structure to improve memory performance, depending on several factors such as what data types are present and how frequently they are used. By applying these optimisations, it can produce benefits such as reduced memory consumption and more efficient cache utilisation.

2.2.2 Array of Structures (AoS)

An array can be created to store struct elements - this is called an *Array of Structures* (AoS). As with any other array, each struct element of the array is stored and accessed contiguously in memory.

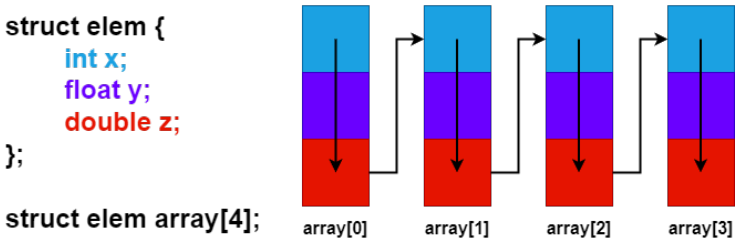


Figure 9: Layout of an AoS in memory [16]

Figure 9 shows the general layout of an AoS in memory, with the arrows indicating the access patterns of each array element. There is good locality within the structure, meaning that the AoS is best used if all the fields of the each structure is accessed sequentially. If fields of different array elements are used, for example accessing `float y` from array[0] and `int x` from array[3], this will require more frequent memory accesses to fetch the 0th and 3rd array elements, which will increase memory latency and degrade performance. Therefore, an AoS is not suitable when two or more structures are required for an operation.

An AoS can be declared statically or dynamically. A static AoS is fixed in size and is declared in stack memory of the program. In C, the standard array declaration syntax is used to create a static array of structures [17], as shown in Figure 9. Alternatively, a dynamic AoS is declared in heap memory using C pointers [18]. This means that the size of dynamic array can be changed during runtime, by allocating and de-allocating memory using `malloc()`/`calloc()` and `free()` function calls respectively [19].

Aside from how the array is declared, the scope of the array can also vary. A local declaration of an AoS means that it is defined inside a function body and can only be used within it. A global declaration of an AoS allows the array to be used by any function within the program, since it is not explicitly defined in

a particular function body. Therefore, there are a total of four variants of AoS data structures shown in Table 1, where each variant is going to be detected and optimised in this project.

Table 1: The four possible AoS variants

Variants of AoS
Local, **static** AoS
Global, **static** AoS
Local, **dynamic** AoS
Global, **dynamic** AoS

2.2.3 Structure of Arrays (SoA)

Instead of storing an array of struct elements, a single structure can be declared to contain arrays of each field type. This is called a *Structure of Arrays* (SoA).

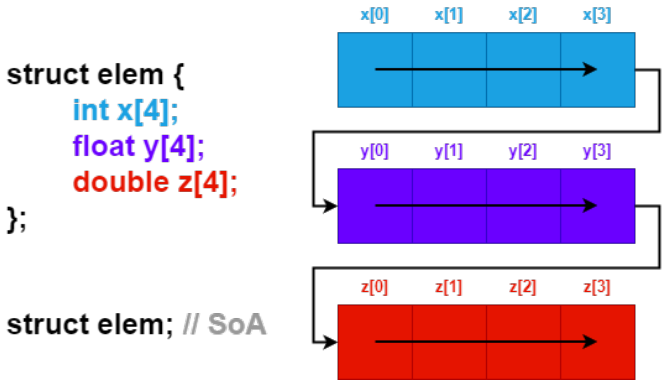


Figure 10: Layout of an SoA in memory[16]

Figure 10 shows the general memory layout of an SoA. The arrows shows that all elements of an array have to be traversed before moving onto the next array field. This means that there is good locality within the arrays, so an SoA is best used for operations that process individual arrays in a contiguous manner. If the program requires two values from two of the array fields, this means that both arrays needs to be fetched to access the data. Depending on the array sizes, this can increase the memory latency and impact performance. Therefore, an SoA is not suitable when elements from two or more different array fields are required for an operation.

Similar to an AoS, there are also four different variants of an SoA, based on its declaration type and its scope shown in Table 2. An SoA can be declared using a struct pointer or using a normal variable declaration for a struct - both types will be referred to as ‘pointer’ and ‘non-pointer’ structures respectively.

The names ‘static’ and ‘dynamic’ is not used because the size of an SoA which is always equal to one so there is no reference to its size. Additionaly, an SoA can have a local scope within a function, or a global scope within the whole program.

Table 2: The four possible SoA variants

Variants of SoA
Local, **non-pointer** SoA
Global, **non-pointer** SoA
Local, **pointer** SoA
Global, **pointer** SoA

2.2.4 Array of Structure of Arrays (AoSoA)

A combination of both AoS and SoA can be used to create a data structure called an *Array of Structure of Arrays* (AoSoA), which is an array that stores SoA data structures as its elements. As shown in Figure 11, this is done to gain the locality benefits of both AoS and SoA; the resulting AoSoA is able to have good locality within the inner most level (arrays) and the outermost level (structs). Although it is difficult to implement and utilise within a program, the AoSoA data structure is most ideal if the access patterns of data are scattered within the structs and arrays. Otherwise, if the access pattern is known, it is best to use an AoS or SoA instead to keep the code simple and reduce memory consumption. An AoSoA would have 16 different variants, depending on how the array itself and each SoA element is declared.

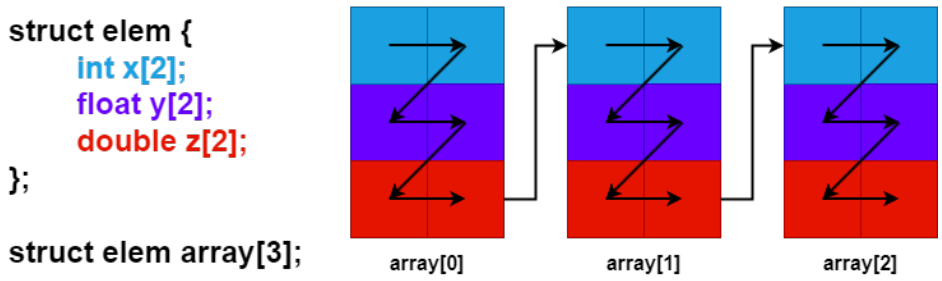


Figure 11: Layout of an AoSoA in memory [16]

2.2.5 Linked Lists

A linked list is an Abstract Data Type (ADT) that is very similar to an array, however each element is not stored contiguously. Figure 12 shows the representation of a singly-linked list data structure. A pointer variable `head` is used to store the first node of the list. Each node contains a pointer which ‘points’ to the

address location of the consecutive node in the list. All of the nodes in the list can be traversed up to the last node, which will not point to a new node indicating the end of the list [15, pp. 163–164].

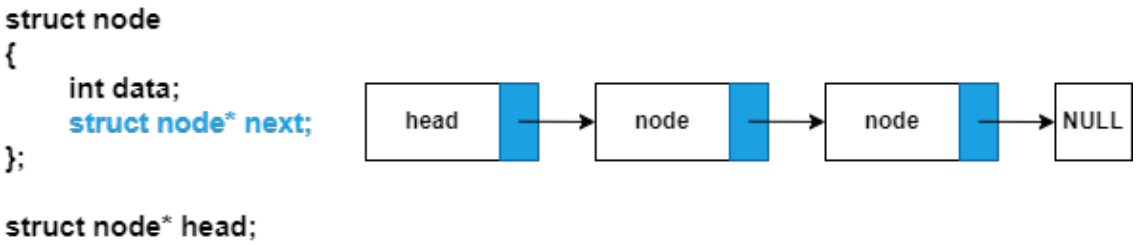


Figure 12: Representation of singly-linked list

In C, each node can be implemented as a struct that should contain a pointer of the same struct type, which is used to reference the next node in the list. Since a linked list can vary in size, it can only be implemented using a pointer declaration. In essence, a linked list in C is very similar to a dynamic AoS; both are pointers which store 'struct' elements. However, each struct element in a linked list must be *recursive*, meaning that the struct contains a pointer to itself [15, p. 163].

2.2.6 Trees

A binary tree is a data structure, consisting of nodes that cascade down in a tree-like structure. The top-most node is called the 'root' node, and each node has at most two 'child' nodes. [15, p. 281].

There are two representations of a binary tree: linked representation and sequential representation. This project focuses on linked representations in C which uses structs to represent each node, that contains two pointers that connect to the left and right 'child' nodes, as shown in Figure 13. Similar to linked-lists, each struct must be recursive so it can reference both 'child' nodes of the same type. This is similar to a linked-list data structure, but it contains two pointers instead of one.

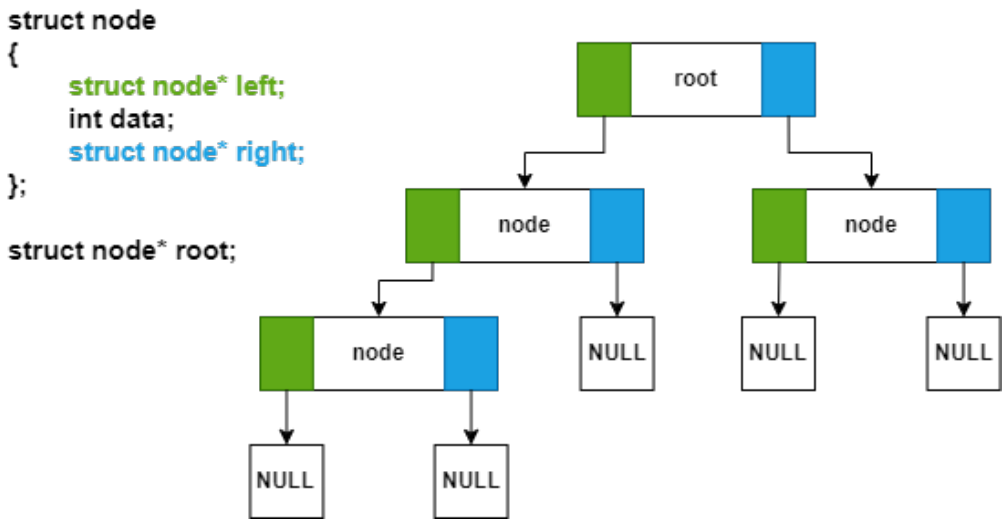


Figure 13: Representation of binary tree [15, p. 283]

2.3 Available Optimisations

2.3.1 Cache memory

The optimisations implemented in this project focus on improving the usage of cache memory during the runtime of a program. Aside from the small-sized registers within the CPU, cache memory is the fastest available memory storage available at the top of the memory hierarchy. Whenever data and instructions are fetched from main memory, a copy of it is also stored in cache, so that the CPU can reference it more quickly if needed again, supporting the principle of *temporal locality* [1, p. 49]. Additionally, caches also support the notion of *spatial locality*, where multiple words of data from main memory are copied into cache since memory addresses referenced close together in time would be likely to be stored close to each other [1, p. 81].

Cache memory is broken down into a sequence of *cache lines*, which are 64 bytes in size as shown in Figure 14. Each cache line consists of 8 *word lines*, that have a size of 8 bytes each. The term 'word' defines the maximum number of bytes the CPU can process in one clock cycle; for a 64-bit processor that is focused on in this project, this is equivalent to 8 bytes of data. This means that one clock cycle is required to process one word line in cache. For the single cache line shown in Figure 14, the word lines are placed vertically for a easier view, but in practice, all eight word lines exist as one row, representing a cache 'line'.

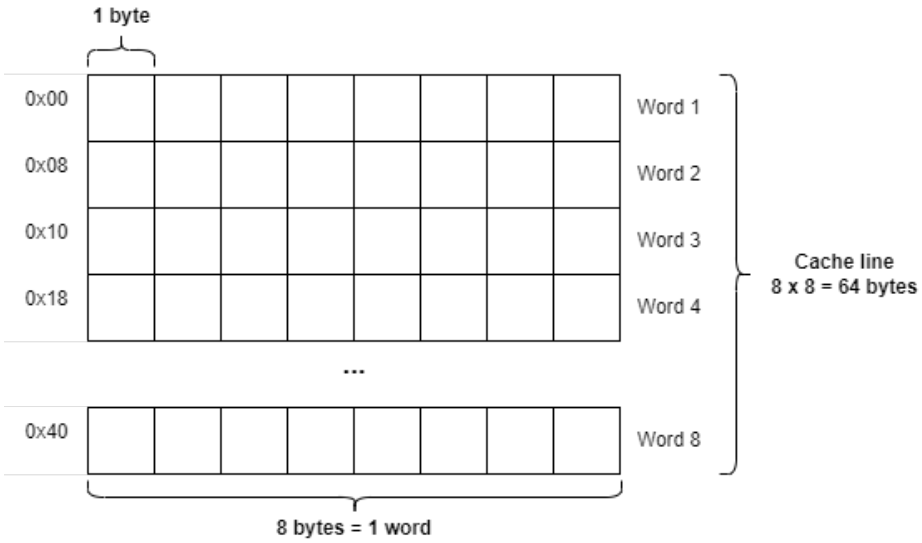


Figure 14: Structure of a cache line

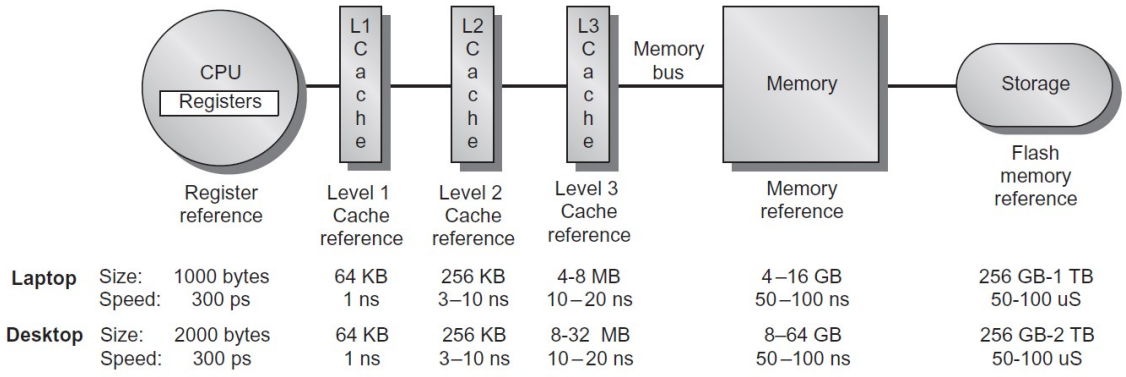


Figure 15: Memory hierarchy for a laptop and desktop computer [1, p. 79]

Looking at the rough sizes and speeds in Figure 15, cache memory can almost have 100 times faster access times than main memory, however its available storage capacity is very limited, due to how expensive it is to implement a larger cache. Therefore, it is beneficial to use more of faster cache memory in order to reduce memory latency and improve memory performance overall. However due to its limited size, it is important to maintain efficient cache utilisation by making memory layout optimisations to the data structures.

For the aforementioned data structures in Section 2.2, the memory layout optimisations will be targeted towards the struct elements. Figure 16 shows a sample struct elements stored in a cache line, which stores each struct field in the exact order they have been defined. This struct is 16 bytes in size so it would take

up two word lines. This means that to fully retrieve and access the struct, two clock cycles are required.

The optimisations implemented in this project aim the change the layout of structs within cache lines. These modifications can lead to the reduction in clock cycles to access the data, which can reduce execution times of programs. Furthermore, by reducing the size of structs, more struct elements of AoS / AoSoA data structures can be stored within cache, which can be fetched faster compared to main memory. By making the program less reliant on slower main memory accesses, the total execution times of programs can be reduced this way too.

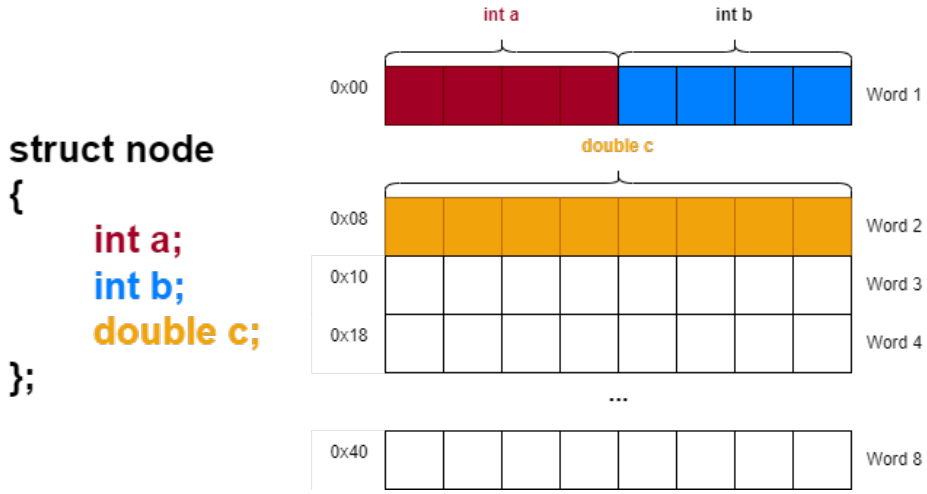


Figure 16: Layout of a struct within a cache line

2.3.2 Structure Field Re-ordering

In C, data types have *alignment requirements*, which represents the number of bytes between successive addresses at which objects of this type can be allocated [20]. Table 3 shows the alignments bytes required for the C data types used in this project.

Table 3: List of development iterations for this project

Data type	Alignment bytes
`char`	1 byte
`int`	4 bytes
`float`	4 bytes
`double`	8 bytes

Alignment requirements are enforced to ensure that the whole data is accessed within one clock cycle.

The struct fields shown previously in Figure 16 are aligned, which means that each field can be accessed with one clock cycle. An example of unaligned struct fields is shown in Figure 17, where the field `double b` is split between two word lines which requires two clocks cycles to fully access the data.

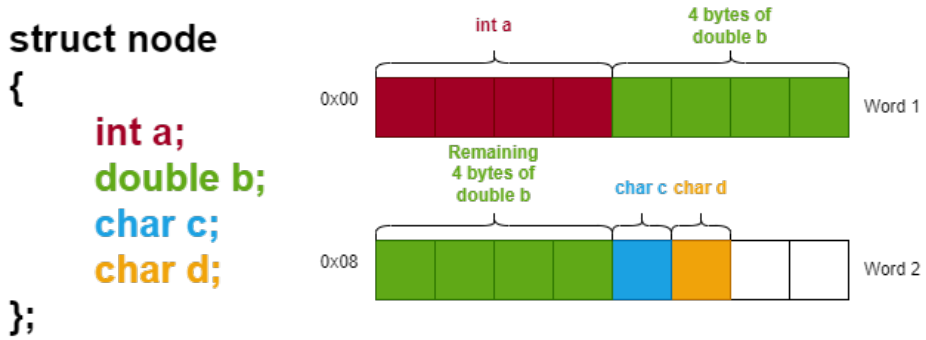


Figure 17: Unaligned struct

To make this field meets its alignment requirements, it can be stored in the next cache line, starting at address `0x08`, by adding *padding bytes* to fill up the first word lines, as shown in Figure 18. Padding refers to empty bytes that are inserted to make sure that all members of a struct satisfy the alignment requirements [20]. The third word line is also filled with padding bytes to separate this struct data structure with the next consecutive data type.

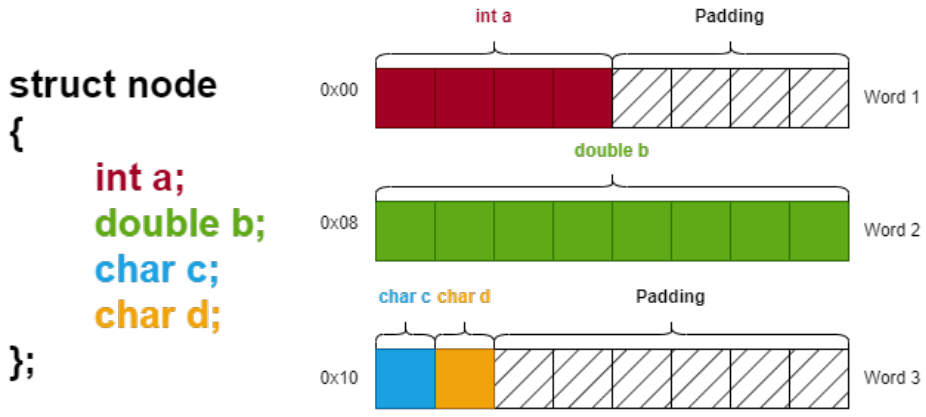


Figure 18: Aligned struct

A disadvantage of padding is that it increases the total size of structs, meaning that more clock cycles would be required to access the whole struct data. Not only this, most of the bytes within the struct may remain unused, since they are used as padding bytes. In the unaligned struct in Figure 17, the total size is 14 bytes, whereas in the aligned struct in Figure 18, the new size is 24 bytes. The increased struct size means that three clock cycles are required to fetch the data, compared to the previous two clock cycles. For repeated accesses to this whole struct, this has a negative impact on execution times despite only requiring one extra clock cycle.

Aligning this struct also increases memory usage. The ten additional bytes (42% of the total size) are used for padding and remain unused within the struct. For a single struct element, this would seem insignificant. However, for large arrays of this struct and multiple copies of this array, large number of bytes would be used for padding, which can be considered wastage since it is unused and could be utilised elsewhere within the program. This would be a problem for heavily memory-dependant programs such as physics simulations, which require most of the available memory to collect and store large amounts of results and calculations. Therefore, it is important to reduce the number of padding bytes used within structs when aligning the fields.

The solution for using the minimal amount of padding bytes within a struct is to perform a basic optimisation called *field re-ordering*. This is the process of re-ordering the fields of a struct, so that the layout of the data fields in memory can be defined. In attempt to reduce the number of padding bytes, the fields can simply be ordered in **decreasing allocation size**. Looking at Table 3, this means that the `double` data fields will always be placed in the beginning and the `char` data fields will be placed at the end.

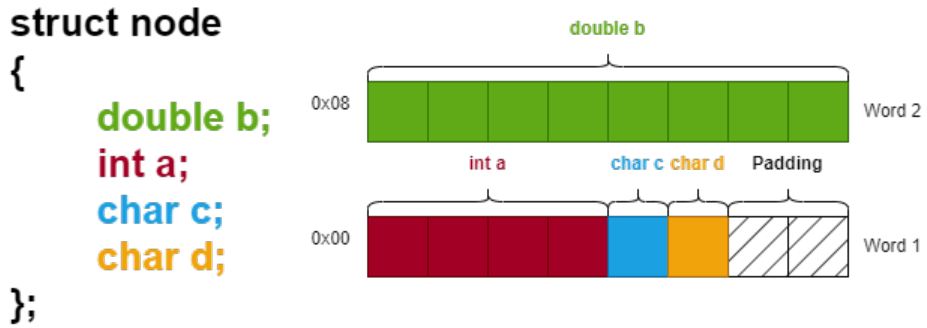


Figure 19: Re-ordering the fields of the struct

Figure 19 shows the struct after re-ordering the fields. The size of this new struct has been reduced by 8 bytes, as it uses less padding bytes. This means that two clock cycles are required to fully fetch the data, and the struct also consumes less memory space. After reordering the fields of a struct, it is possible that gaps may be present between fields, which are taken up by padding bytes. These padding bytes can be removed by checking whether any of the fields of the struct is able to fit within the padding space and then inserting the field into that space. This is a refinement to the optimisation that could further reduce the size of the struct. After reordering the fields, modifications such as *peeling* and *splitting* can be applied to further optimise the struct.

2.3.3 Structure Peeling

Certain fields within a struct may be used more frequently than others. This would means that the struct is partially used within the program, so most of the bytes of an fetched struct is essentially unused within parts of the program. The most frequently used fields can be separated from the least used fields by creating two separate structs for the fields, which would help reduce the struct size. This would provide performance benefits in terms of memory usage and execution times, since less clock cycles would be required to process the smaller structs.

This process is called *struct peeling*, which 'peels' the original struct into a 'hot' struct and a 'cold' struct which stores the most frequently-used fields and least used fields respectively. The measure of how frequent each fields has been used within a program is referred to as *hotness*.

To measure hotness of each field of a struct, the program code is analysed and the number of uses for each field is counted. Inside loops, the counts within each iterations can also be considered for a more accurate hotness value. There are several ways to separate the fields of the struct based on its hotness values, such as using a hotness threshold value or including other metrics such as *affinity*, which measures how close the fields are accessed together. This project does not explore the effectiveness of all of these methods and it only uses the simplest method: calculating the **mean** hotness value and separating them to 'hot' or 'cold' structs if the value is greater or equal to or less than the mean value respectively.

After peeling, the array that contained the original struct elements can now store the 'hot' structs, whilst a new array must be created to store the 'cold' structs. Lastly, the cold field accesses must be updated so that it now accesses the cold structs from the new cold array.

```c
struct nodeOne
{
    double a; // hotness = 5000
    double b; // hotness = 2000
    double c; // hotness = 2000
    int d; // hotness = 1000
    int e; // hotness = 1000
    char f; // hotness = 0
};

// mean hotness = 1833

// unoptimised data structure
struct nodeOne aos[1000];

int main()
{
    for(int i = 0; i < 1000; i++)
    {
        aos[i].a = 9934.2;
        aos[i].b = 1234.43;
        aos[i].c = 50.23;
        aos[i].d = 100;
        aos[i].e = 100;
    }

    for(int i = 0; i < 1000; i++)
    {
        aos[i].a = aos[i].a *
        ↪  aos[i].a;
        aos[i].b = aos[i].c +
        ↪  aos[i].a;
    }
}
```

Listing 5: Before struct peeling

```c
struct nodeOne // 'hot' struct
{
    double a;
    double b;
    double c;
};

struct nodeOneCold // 'cold' struct
{
    int d;
    int e;
    char f;
};

// optimised data structure
struct nodeOne aos[1000];
struct nodeOneCold aosCold[1000];

int main()
{
    // modified 'for' loop
    for(int i = 0; i < 1000; i++)
    {
        aos[i].a = 9934.2;
        aos[i].b = 1234.43;
        aos[i].c = 50.23;
        aosCold[i].d = 100;
        aosCold[i].e = 100;
    }

    // same second 'for' loop as
    ↪  before
}
```

Listing 6: After struct peeling

Figure 20: Comparison of C code before and after applying structure peeling

Figure 20 shows a comparison between the code, before and after applying the structure peeling optimisation. From Listing 5, the first three fields have hotness values greater than the mean, so it is put into the 'hot' struct whilst the remaining fields are placed in the 'cold' struct. In Listing 6, a new array is created in Line 16 to store the 'cold' structs and since the first for loop uses the 'cold' fields at Lines 25-27, these accesses must be updated to use the new 'cold' struct instead. By performing this optimisation, the original struct size has been reduced by 16 bytes. This is beneficial for the second `for` loop in the

code, since it only accesses the top three fields of the original struct. By making it access the 'hot' struct instead, it only requires three clock cycles to retrieve the required fields, instead of five clock cycles. Additionally, the `for` loop will benefit from spatial locality of reference, since the required fields will be stored together in cache. These two benefits will ultimately result in a reduction of execution times.

Despite the advantages of this optimisation, there are limitations on the types of structs that can be peeled. Firstly, recursive structs cannot be peeled. Recursive structs contains a pointer fields to itself, so peeling the struct is not ideal since two new structs are created and the struct pointer field will no longer reference either of these structs. Some code may use this struct pointer to access both a 'hot' and 'cold' field of the original struct, which would no longer exist together after peeling. On the topic of recursive structs, it is very difficult to identify one in LLVM IR, due to pointers having no associated types as they are referred to as *opaque pointers* [21]. It would make it impossible to determine the type of a pointer fields inside the IR because they are simply represented as `ptr`, hence for this project, all structs that contain `ptr` fields are classified as recursive structs. Therefore in this project, structure peeling is **not** applied to structs containing pointer fields.

Secondly, structure peeling cannot be applied to structs that are used as function arguments. This is because a function may take in the original struct as an argument and use both 'hot' and 'cold' fields. After peeling, this struct argument will no longer contain the required 'hot' and 'cold' fields so its needs to take in the 'cold' struct as a new function argument too. For these scenarios, it requires modifications to the function prototype, which would be difficult to do in LLVM IR especially with the high risk of breaking the program. Therefore, considering this limitations, structure peeling is applied to all global AoS and local, static AoS declarations, as it is safe to apply the optimisations to these data structures without possibly breaking the program's functionality. Local, dynamic AoS declarations and the other variants are optimised using structure splitting instead.

2.3.4 Structure Splitting

Structure splitting is very similar to structure peeling, in the sense that its identifies the 'hot' and 'cold' fields and separates it into 'hot' and 'cold' structs. The only difference here is that a new array is not created to store each 'cold' struct individually, instead the 'hot' struct will include a pointer field that will reference the 'cold' struct for each array element. Even though it introduces an overhead to accessing the cold fields via pointer dereferencing, this optimisation can be applied to a wide range of structs without

any major constraints. like seen with structure peeling. Figure 21 shows an example of applying this optimisation to the same C code seen with structure peeling.

```c
struct nodeOne
{
    double a; // hotness = 5000
    double b; // hotness = 2000
    double c; // hotness = 2000
    int d; // hotness = 1000
    int e; // hotness = 1000
    char f; // hotness = 0
};

// mean hotness = 1833

int main()
{
    // unoptimised data structure
    struct nodeOne aos[1000];

    for(int i = 0; i < 1000; i++)
    {
        aos[i].a = 9934.2;
        aos[i].b = 1234.43;
        aos[i].c = 50.23;
        aos[i].d = 100;
        aos[i].e = 100;
    }

    for(int i = 0; i < 1000; i++)
    {
        aos[i].a = aos[i].a *
        ↪ aos[i].a;
        aos[i].b = aos[i].c +
        ↪ aos[i].a;
    }
}
```

Listing 7: Before struct splitting

```c
struct nodeOneCold // 'cold' struct
{
    int d;
    int e;
    char f;
};

struct nodeOne // 'hot' struct
{
    double a;
    double b;
    double c;
    // pointer to 'cold' struct
    struct nodeOneCold* cold;
};

int main()
{
    // optimised data structure
    struct nodeOne aos[1000];

    // modified 'for' loop
    for(int i = 0; i < 1000; i++)
    {
        aos[i].a = 9934.2;
        aos[i].b = 1234.43;
        aos[i].c = 50.23;
        // allocate memory to 'cold'
        ↪ struct ptr
        aos[i].cold = (struct
        ↪ nodeOne*)
        ↪ malloc(sizeof(struct
        ↪ nodeOneCold));
        aos[i].cold->d = 100;
        aos[i].cold->e = 100;
    }

    // same second 'for' loop as
    ↪ before
}
```

Listing 8: After struct splitting

Figure 21: Comparison of C code before and after applying structure splitting

Since the ‘cold’ struct pointer is present inside the ‘hot’ struct, seen at Line 14 of Listing 8, there is no need to create a new array to store the ‘cold’ struct elements. Additionally for every ‘cold’ field access within the code, this ‘cold’ struct pointer must be allocated memory dynamically, as shown in Line 28 of Listing 8. Performing this optimisation provides the same performance benefits as structure peeling; the structure size has been reduced and good spatial locality is seen.

It could be said that struct splitting is more superior to struct peeling, since it can be applied to all types structs, without any constraints. However, this optimisation is not as effective as struct peeling, due to the overhead caused by introducing pointers and the need to reference and dereference from it [22]. Ultimately, it was decided to utilise both optimisations so the benefits of struct peeling can be seen as well as being able to make optimisations available to all types of AoS data structures using struct splitting.

2.3.5 AoS to SoA Conversion

As detailed previously, an AoS is best used if all the struct fields of an array elements are accessed sequentially. Some programs may access only one field of each AoS element in non-sequential manner, meaning that the AoS is not used efficiently as each struct has to be loaded in and out for only one field. Using an SoA here would be more ideal, since each field could be accessed sequentially using an array. Therefore, conversions from AoS to SoA data structures also provide simple optimisations that can boost memory performance. Additionally, converting to SoA data structures allows for new compute-bound optimisations to be applied to the code, such as loop vectorisation and loop unrolling which allows multiple loop iterations to be processed at once in order to reduce execution time [23].

2.3.6 AoS to AoSoA Conversion

Some programs may traverse through each struct element in an AoS and also access multiple fields of the structs, so converting to an SoA data structure is not ideal here since it reduces the locality between accesses to multiple fields [23]. In this case, converting to an AoSoA data structure can provide the benefits of both AoS and SoA: good locality within the struct elements and the array fields. Despite the difficulty of implementing an AoSoA from an AoS, it can provide the best reductions in execution times.

3 Design and Implementation

As shown previously in Figure 5, the compiler passes for optimisation and detection is run in sequence within the pass pipeline of the optimiser. A top-level compiler pass, appropriately called `allPasses`, will be used to run all compiler passes on the target program. This pass uses the code in Listing 9 to initiate the pass manager, which is responsible for running each compiler pass in a specific order.

Listing 9 Pass manager inside the 'allPasses' compiler pass.

```
extern "C" ::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_WEAK
llvmGetPassPluginInfo() {
  return {
    LLVM_PLUGIN_API_VERSION, "allPasses", "v0.1",
    [](PassBuilder &PB) {
      PB.registerPipelineParsingCallback(
        [](StringRef Name, ModulePassManager &MPM,
        ArrayRef<PassBuilder::PipelineElement>) {

        // Code for the management of compiler passes

        }
      );
    }
  };
}
```

Listing 10 General format for using the **opt** command to run a compiler pass

```
opt -load-pass-plugin=./allPasses.so -passes="passName" < unoptimised.bc >
↪  optimised.bc
```

Once this pass is compiled as a `.so` file, it will be used as a pass plugin that is run using a LLVM command on a bash terminal shown in Listing 10. Within this command, an additional argument `passName` is required to specify the single pass or combination of passes to be run within `allPasses`. The code inside `allPasses` contains `if` statements that check the `passName` argument and run the appropriate passes in the pipeline.

Table 4 shows the possible string values that can be given to the `passName` argument. The first eight pass combinations are mainly used in this project for testing purposes and for detailed discussion in this report. The last four pass combinations are present to show that the compiler passes can be applied together in any order, without causing unexpected errors and breaking the optimised program.

Table 4: List of available pass combinations to run in pass pipeline

Type	`passName` **value**	**Description**
Detection ONLY	**detectSoA**	Detection of SoA
	detectAoSoA	Detection SoA, AoS and AoSoA
Detection & SINGLE Optimisation	**reorderAoS**	Reorder fields of AoS structs
	reorderStructs	Reorder fields of all structs
	peelAoS	Apply structure peeling to AoS data structures
	splitAoS	Apply structure splitting to AoS data structures
Detection & MULTIPLE Optimisations	**peel+reorderAoS**	Apply structure peeling, then structure field reordering to AoS data structures
	split+reorderAoS	Apply structure splitting, then structure field reordering to AoS data structures
	reorder+peelAoS	Apply structure field reordering, then structure peeling to AoS data structures
	reorder+splitAoS	Apply structure field reordering, then structure splitting to AoS data structures
	peel+splitAoS	Apply structure peeling, then structure splitting to AoS data structures
	split+peelAoS	Apply structure splitting, then structure peeling to AoS data structures

An additional compiler pass not shown in the above table is run before any detection and optimistion passes are applied. This is called `removeConstantGEP`, which removes all **constant** `GetElementPtr` (GEP) instructions within the IR of the unoptimised program. `GetElementPtr` instructions are heavily used in the detection and optimisation processes, as they are used to access arrays and structs [24]. Any constant variants of this instructions needs to be converted to a standalone GEP instruction, otherwise they cannot be easily detected and modified by the optimisation passes. This will be problematic after certain optimisations, such as structure field reordering, where the operands of GEP instructions that rep-

resent field accesses may need to be updated. A constant GEP cannot have its operands updated, so not updating these can cause the optimised program to break with errors such as segmentation faults or produce unexpected results. The generation from source code to LLVM IR can automatically create constant GEP instructions to simplify the IR where necessary, and there is no way to disable it. Therefore, it is important to remove constant GEP instructions before optimising, to make sure all GEP instructions are available for detection and modification so the program's functionality is not broken after optimisation.

All detection passes make use of 'potential' and 'confirmed' vector containers, that is defined inside the shared `detectAoS.h` header file. If a data structure within the IR matches a certain criteria, that will be discussed in detail in this section, the data structure, in the form of a `Value` instance, will be added to 'potential'. Detection does not conclude here since these AoS data structures need to be used within the program to be counted as a 'confirmed' AoS data structure, in order for it to be flagged for optimisation. This means that unused AoS data structures are not optimised, and this decision was made because optimising unused data structures would be a wasted effort, since there would not be any performance benefits. Instead of optimising these and having the risk of breaking the program functionality, it is best to not touch unused data structures in the program. There also additional reasons why an AoS must have a data access in order to be correctly detected, especially for dynamic AoS data structures that is explained in the following sections.

3.1 Detection of AoS

The compiler pass `detectAoS`, is an analysis pass that detects both static and dynamic AoS data structures, without modifying the input IR. It is broken down into three components shown in Figure 22. Each of the three stages of this compiler pass will be discussed in detail within this section.

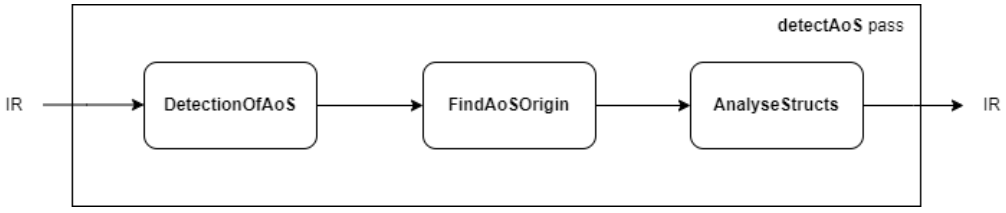


Figure 22: General structure of 'detectAoS' compiler pass

The AoS data structures are found during the 'DetectionOfAoS' phase, where they are stored in the 'confirmed' vectors to be used by the optimisation passes. Figure 23 shows the top-view design of the detection phase.

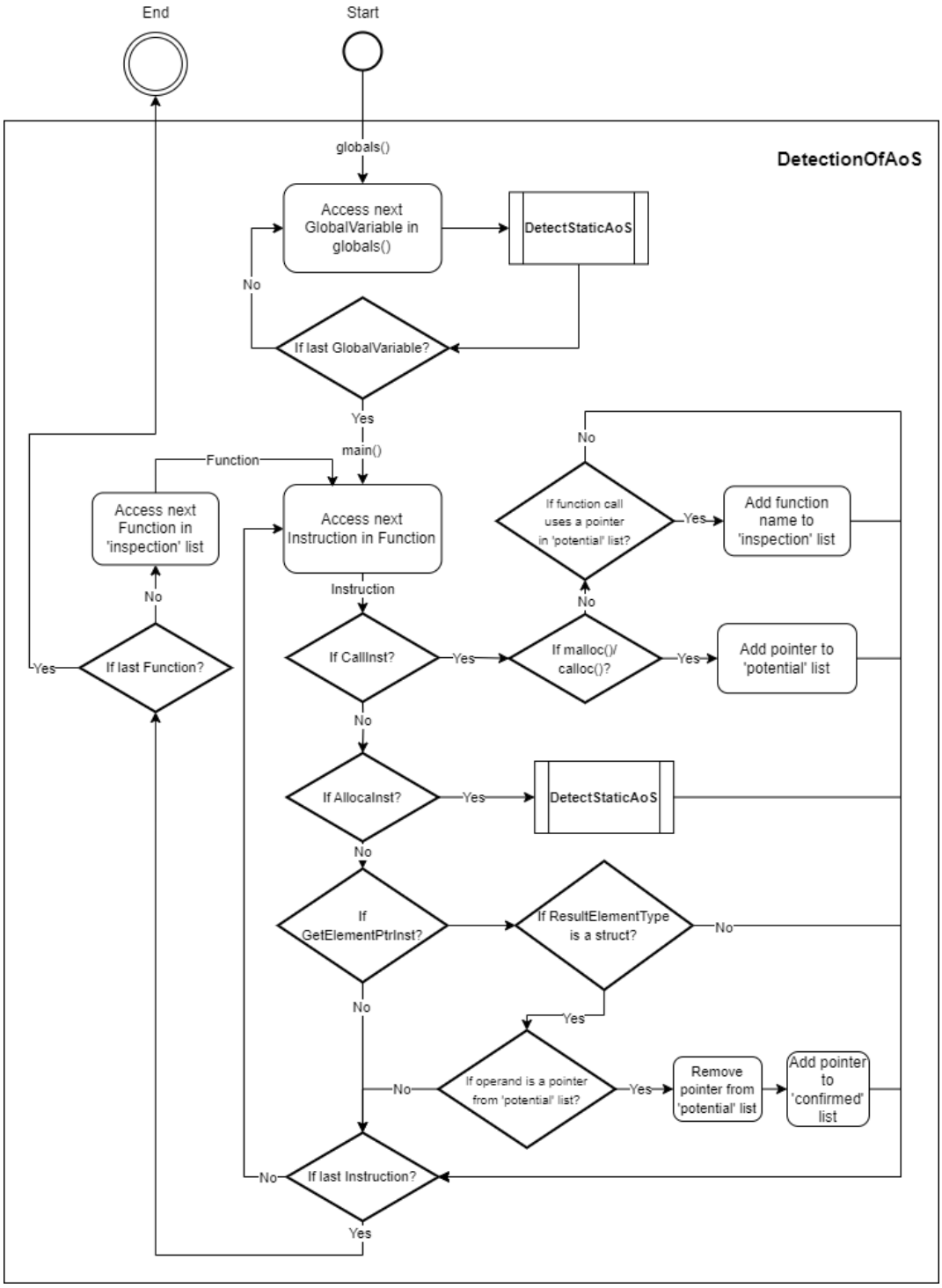


Figure 23: Top-view design of the AoS detection phase in the ‘detectAoS’ compiler pass

During detection, the compiler pass first checks through all of the global variables to look for any static AoS declarations, which are put into the ‘potential’ vector so a data access from it can be found later. After this is performed, the main() function block is analysed to look for any locally-declared, static and

dynamic AoS data structures. This function block may also contain function calls, with zero or more arguments. The function arguments could be AoS data structures that are already stored in 'potential' and are waiting for a data access to be found. Therefore, it is important to store the function names of these function calls in a vector called 'inspection' so that it can be later inspected. Alternatively, these function arguments may be new global pointer variables that could be allocated heap memory (using 'malloc()' or 'calloc()' function calls) and used as an AoS data structures within the function calls, so it is also important to add these global variables to a new vector 'possibleGlobals'.

Once every instruction inside the main() function is checked, the compiler pass proceeds to inspect every called function inside the 'inspection' vector. The same detection process that happened for the main() function occurs for each function in the vector 'inspection' until there are no functions to analyse, indicating the end of the AoS detection phase.

Figure 24 shows the top-view design of the detection of static AoS declarations. This can either be local or global, depending on whether an `AllocaInst` or a `GlobalVariable` is being analysed respectively.

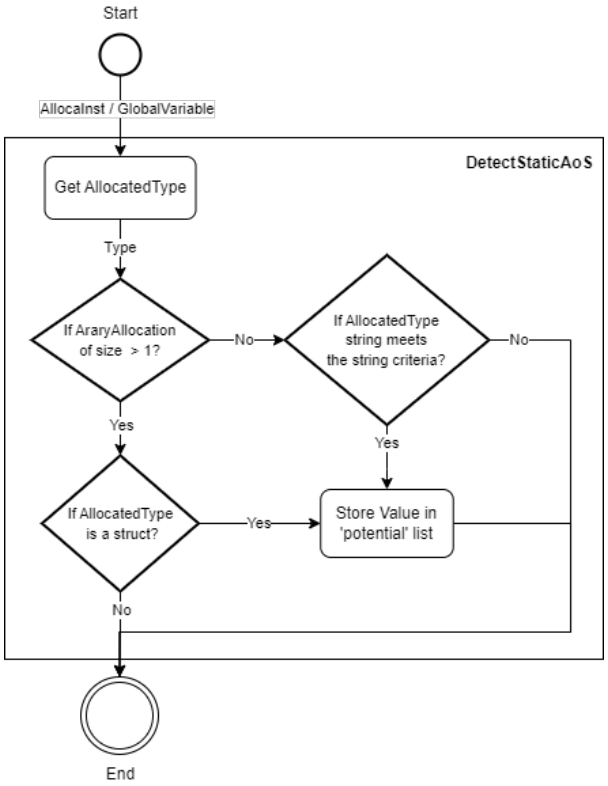


Figure 24: Top-view design of the static AoS detection

In terms of the IR, a static AoS is represented as an `alloca` instruction shown in Listing 12, which is represented by the `AllocaInst` class to allocate memory on the stack [25]. From this class, a function `getAllocatedType()` is available, which returns the type that is being allocated by the instruction. This type should be an `ArrayType`, which represents array types in IR [26]. Within this class, an function called `getElementType()` is available, which should provide the type of each array element. The element type should be a `StructType`, to indicate that struct elements are stored. If both types are correct, an additional check is required to check where the declared `ArrayType` has a size greater than one. A static AoS is only detected if it has more than one struct element, otherwise it is just a single struct element that cannot really be classified as an array. This is checked using the `getNumElements()` function that is available within the `ArrayType` class [26]. If these three checks match for a particular `alloca` instruction, a static AoS declaration has been found within a function body.

```
1  struct node {
2  int a;
3  float b;
4  double c;
5  char d;
6  };
7
8  int main() {
9  // ...
10 struct node
   ↪  arrayOne[100];
11 // ...
12 }
```

Listing 11: C representation

```
1  %struct.node = type { i32, float, double, i8 }
2
3  ...
4
5
6  ...
7
8  define dso_local i32 @main() #0 {
9  entry:
10 ...
11 %arrayOne = alloca [100 x %struct.node], align 16
12 ...
13 }
```

Listing 12: IR representation

Figure 25: C implementation and IR representation of the static AoS detection within the main() function

C declarations of a global static AoS in Listing 13 are represented using `GlobalVariable` instances in the IR, which is created outside of function bodies as shown in Listing 14. To detect a global, static AoS, each `GlobalVariable` is checked by iterating through the `globals()` list provided by the top-level `Module` class [27]. For each `GlobalVariable`, the initializer is retrieved using the function `getInitializer()`, which returns the initial value set to the global variable as a `Constant` [28]. The method `getType()` inherited from the `Value` class can be used on this `Constant` to determine whether it is an ArrayType. After converting the Constant instance to an ArrayType instance, a special function `getArrayElementType()` from the `Type` class can be used to easily determine the type of each array element [29]. Finally, this

element type is checked to see if it is a `StructType`, and if so, a global static AoS has been found.

```
struct nodeOne {
char a;
float b;
int c;
double* d;
};

struct nodeOne
    ↪  arrayOne[917];

struct nodeTwo {
int a;
float b;
double c;
char d;
} arrayTwo[521];
```

Listing 13: C representation

```
%struct.nodeOne = type { i8, float, i32, ptr }

%struct.nodeTwo = type { i32, float, double, i8 }


...


@arrayOne = dso_local global [917 x %struct.nodeOne]
    ↪  zeroinitializer, align 16

@arrayTwo = dso_local global [521 x %struct.nodeTwo]
    ↪  zeroinitializer, align 16

...
```

Listing 14: IR representation

Figure 26: C implementation and IR representation of the static AoS detection within the main() function

When performing the detection of data structures, it may be necessary to convert between one class instance to another, in order to access its functions. This is done using a `dyn_cast<>` operator, which performs a "checking cast": it first checks whether the operand is of a specified type, and if so, returns a pointer to that type by casting to it.[30]. This operator is used several times within the detection and compiler pass and an example of this is shown in Listing 15, which is used to check whether the initalizer type of a `GlobalVariable` is equal to an `ArrayType`.

Listing 15 Example usage of `dyn_cast<>` operator in C++ API

```
if(auto *ArrTy = dyn_cast<ArrayType>(initializer_type)){
   // Type inializer_type is converted to an ArrayType pointer *ArrTy
}
```

Dynamic AoS data structures, both local and global declarations, are represented as pointers in IR, which have been allocated memory using `malloc()` or `calloc()` function calls as shown in Listing 17. Firstly, function calls need to be detected, which are represented as `call` instructions using the `CallInst` class. As seen in Listing 17, each `call` instruction contains the called function names, which in this case is either `malloc` or `calloc`. The returning types of these function calls are pointers, as seen by the usage of the term `ptr` in the IR.

Listing 16 C representation of dynamic AoS data structures

```
1  struct nodeOne {
2      char a;
3      float b;
4      int c;
5      double* d;
6  };
7
8  struct nodeOne* arrayGlobal; // Global, dynamic AoS pointer
9
10 int main() {
11     struct nodeOne* arrayLocal; // Local, dynamic AoS pointer
12
13     arrayLocal = (struct nodeOne*) malloc(914 * sizeof(struct nodeOne));
14     arrayGlobal = (struct nodeOne*) calloc(750, sizeof(struct nodeOne));
15 }
```

Listing 17 IR representation of dynamic AoS data structures

```
1  %struct.nodeOne = type { i8, float, i32, ptr }
2
3  @arrayOne = dso_local global ptr null, align 8
4
5  define dso_local i32 @main() #0 {
6  entry:
7  %arrayLocal = alloca ptr, align 8
8  %call = call noalias ptr @malloc(i64 noundef 21936)
9  store ptr %call, ptr %arrayLocal, align 8
10 %call = call noalias ptr @calloc(i64 noundef 750, i64 noundef 24)
11 store ptr %call, ptr @arrayGlobal, align 8
12 }
```

In C, pointers can have a type (e.g. `int*` is an integer pointer), so it is easier to detect a struct pointer by inspecting the code, such as lines 8 and 11 in Listing 16. This means that a dynamic AoS can be simply detected within the C source code by identifying a struct pointer. This method, however, cannot be applied to pointers in IR. As specified previously, this is because LLVM IR uses *opaque pointers*, which do not use types, such as `i32*` for `int` pointers. Old versions of LLVM used typed pointers, with available functions in the API to obtain the pointer types. For newer versions of LLVM, these pointers slowly transitioned to opaque pointers, due to a lack of pointee type semantics and other type-related issues, meaning that the functions to access the types of pointers were deprecated [21]. Therefore, an alternative method is required to determine whether a pointer in IR represents a dynamic AoS.

One solution is to look at the allocation size given to the `malloc()` or `calloc()` function calls. From

a `calloc()` call, the first operands specifies the total number of objects (of a particular type) and the second operand represents the size of each object [31]. This element size from the second operand can be used to compare with the size of a struct, and if they match, it can be concluded that the `ptr` can store struct elements. A `malloc()` function call uses one operand to represent the total allocation size in bytes [32]. This can be divided by each struct within the program, to see if it is a perfect divisor, and if so, it also means that the pointer can store multiple struct elements. A problem with this solution is that it is not a reliable way to detect AoS data structures. This is because there are various data types that can be stored within the allocation sizes given to these function calls. The `int` and `double` data types are 4 and 8 bytes in size respectively, and these also divide perfectly by the allocation size given to the `malloc()` function call in line 8 of Listing 17, meaning that the `%arrayLocal` pointer can store 5484 `int` elements or 2742 `double` elements, instead of struct elements, therefore there is no conclusive way to determine whether this pointer is an AoS.

The only reliable method available for AoS detection is to look for data accesses of that pointer within the IR. A data access (either a read or write operation) on a struct / array pointer is represented by the `GetElementPtr` (GEP) instructions, which is defined by the `GetElementPtrInst` class [24]. This class provides the useful `getReturnElementType()` function that can be applied to GEP instructions in order to determine whether the access pointer returns a struct. If the returned type is a `StructType`, the accessed pointer indicates an Array of Structures (AoS) data structure with confidence.

This is another reason for the existence of the 'potential' and 'confirmed' vectors for the detection passes. The 'potential' vectors stores all static AoS data structures and potential pointer variables that could be dynamic AoS data structures. If a GEP instruction is found to access a struct from a pointer, that pointer operand is searched inside 'potential' vector, removed from it then added to 'confirmed' vector to indicate that this AoS has been successfully found and optimisations can be applied to it with confidence. Static AoS data structures do not need to have a GEP instruction searched, since the `AllocaInst` or `GlobalVariable` instance provides enough information to confidently detect a local or global static AoS respectively. However, they are not put into the 'confirmed' list immediately, instead they are placed into the 'potential' list so a data access can be found for it. This decision was made because it makes the detection process consistent for all AoS types and it ensures that the static AoS is actually used within the program before time and resources are spent on it for optimisation, otherwise the applied optimisation are not worthwhile given the risks of breaking the program's functionality.

The functions `getSourceElementType()` and `getResultElementType()` are predominantly used within the detection and optimisation passes to determine what is being accessed and what is returned by the instruction respectively [24]. To determine whether a pointer is an AoS data structure, a GEP instruction must access a `ArrayType` and return a `StructType`. So the 'detectAoS' pass checks whether `getSourceElementType() == ArrayType` **and** `getResultElementType() == StructType` in order to confirm that the pointer is a dynamic AoS data structure. Listing 18 shows an example of a value assignment operation to a struct field of an AoS in LLVM IR. The second GEP in line 13 is used to access a field of the returned struct from the GEP instruction in line 12. Only this first GEP instruction (line 12) is required to check whether a struct elements is being accessed in the AoS. Once checked, the instruction of pointer operand `ptr %0` can be retrieved by using the function `getPointerOperand()` from the `GetElementPtrInst` class [24]. This pointer operand would always lead to a `load` instruction, shown in line 11. Since this instruction uses the `LoadInst` class, the loaded pointer, which is the dynamic AoS data structure, can be retrieved using the function `getPointerOperand()` [33]. This pointer is searched and removed from the 'potential' list, to avoid searching for the same pointer again in another data access operation, and the pointer is added to the 'confirmed' list with the relevant information about it, such as name, type and struct used, which helps with applying the optimisations later.

Listing 18 Example of a data access of an AoS struct field in LLVM IR

```
1  %struct.nodeOne = type { i8, float, i32, ptr }
2
3  @arrayFour = dso_local global ptr null, align 8
4  @arrayFive = dso_local global ptr null, align 8
5
6  define dso_local i32 @main() {
7  entry:
8
9  ...
10
11 %0 = load ptr, ptr %arrayOne, align 8
12 %arrayidx = getelementptr inbounds %struct.nodeOne, ptr %0, i64 34
13 %a = getelementptr inbounds %struct.nodeOne, ptr %arrayidx, i32 0, i32 0
14 store i8 99, ptr %a, align 8
15 %1 = load ptr, ptr %arrayThree, align 8
16 call void @populateArray(ptr noundef %1)
17 ...
18 }
```

Once all the static and dynamic AoS data structures are found in the whole program, the second phase of the 'detectAoS' pass, shown in Figure 22, is responsible for finding the correct, original information

for each detected AoS data structure in the 'confirmed' list. Due to the implementation of the detection, AoS data structures that are found within single or multiple depths of function calls do not retain the original information about it, such as the names and the function where it is declared. The recovery of this information is allocated to this second phase of the compiler pass, instead of retaining all of this information during the detection phase, which requires a lot of data structures in the compiler pass to store the information and makes the code more complex.

The 'confirmed' vector stores a tuple of values and what each values represents shown in Listing 19. The value for the second field 'origin function' and first field 'AoS' may not be correct when identifying AoS data structures within function calls. Therefore, these are updated by using the current `Function` value in the second field. A function call for this `Function` value is searched within all function bodies, and if found within another `Function`, a new function call for this function is searched in a similar manner. This process occurs recursively until there are no function calls found, so the `Function` that contains the final function calls becomes the new 'origin function' of the AoS. From this function, the original name can also be fetched and stored in the 'AoS' field, since AoS data structures using as function arguments will have different names that what was used for the declaration.

Listing 19 Example of inserting struct information into the 'originalStructSizes' vector.

```
/*
Tuple elements of each confirmed AoS:
  1 - AoS
  2 - origin function
  3 - "static" or "dynamic"
  4 - struct used
  5 - if it used as a function argument - to determine whether struct peeling or
   ↪  splitting should be applied
  6 - if the struct is recursive - to determine whether struct peeling or
   ↪  splitting should be applied
  7 - AoS type in string format - "AoS" or "AoSoA"
*/
vector<tuple<Value*,Function*,string,StructType*,bool,bool,string>> confirmed
```

With all the correct AoS informations stored in the 'confirmed' vector, the last phase of the compiler pass is to analyse and collect details about the structs used for each AoS. This phase is responsible for storing the original sizes of each AoS struct, and detect any structs that contains pointer fields.

Firstly, the original sizes of each struct is stored in a new vector called 'originalStructSizes', as shown in Listing 20. Each element in this vector consists of a pair of values, with the first value being a `struct`

used by an AoS and the second value being another pair (`origSize, newSize`). As this vector is defined within the `detectAoS.h` header file, this is used by the optimisation passes to compare the new optimised struct size with the original sizes in this vector, in order to determine whether an optimisation is worthwhile in terms of memory reduction. Within this 'detectAoS' pass, both `origSize` and `newSize` are set to the original struct sizes. After an optimisation has been applied within an optimisation pass, the `newSize` value within the inner pair will be updated, to ensure that other optimisations also see the updated struct size.

Listing 20 Example of inserting struct information into the 'originalStructSizes' vector.

```
originalStructSizes.insert(make_pair(struct,make_pair(origSize,newSize)));
```

Secondly, each struct used by the AoS data structures is checked for any pointer operands. This is done by checking if type of each struct element is a `StructType`. All the fields from the struct can be obtained using the function `elements()` from the `StructType` class, which provides them in an iteratable container called `ArrayRef<>` [34]. This enables the usage of a `for` loop to iterate through each struct field, as shown in Listing 21. This for loop exists within another 'for' loop that iterates through each AoS in the 'confirmed' vector, so when a pointer field is found in the current AoS struct, the current element in the 'confirmed' vector is updated with a flag to signify that the used struct contains a pointer field, as shown in line 10.

Listing 21 C++ code for checking each field of a struct

```
1  ArrayRef<Type*> elemArr = structure->elements(); // get all struct fields
2
3  for(auto it = elemArr.begin(); it != elemArr.end(); it++)
4  {
5      Type* ty = const_cast<Type*>(*it);
6
7      if(ty->isPointerTy())
8      {
9          errs()<<"Pointer found within struct\n"; // print to terminal
10         get<5>(confirmed.at(i)) = true; // set correct flag in `confirmed' vector
11         break; // no need to check other fields once ptr found
12
13     }
14 }
```

The list of these structs are later used to ensure that structure peeling is not applied to these structs, since it is not a safe optimisation for these type of structs.

At the end of the 'detectAoS' pass, the 'confirmed' vector has been correctly updated such that any following optimisation passes in the pipeline know what data structures are available to apply optimisations to.

3.2 Implementation of Structure Field Re-ordering

The compiler pass 'reorderAoS' modifies the input IR to apply the struct field reordering optimisation to structs. Despite the given name to this pass, it is implemented to optimise structs used by AoS data structures, **or** optimise all structs within the program, depending on whether the 'detectAoS' pass has been previously run in the pass pipeline. This is determined using a a boolean variable `detectAoS` that is declared in the `detectAoS.h` header file as shown in Listing 22. If the 'detectAoS' pass has been run, this flag is set to `true`, otherwise it remains at `false`.

Additionally, any cold structs may have been created by the struct peeling and splitting optimisations and are possibly not used by an AoS, therefore it may not be targetted for optimisation. Therefore, an additional vector `coldStructs`, shown in Listing 22, is used to store these cold structs so the 'reorderAoS' pass can later append them to the vector called `allStructs`.

Listing 22 Important variables declared in the 'detectAoS.h' header file.

```
// flag to check if all structs or only AoS should be optimised
bool detectAoSCalled;

// used by 'reorderAoS' to ensure cold structs are not skipped for optimisation.
vector<StructType*> coldStructs;
```

Each `StructType` inside the `allStructs` vector will be iterated through and have its fields reordered, if necessary. Figure 27 shows the overall structure of this compiler pass. For each struct to be optimised, the fields are reordered based on its size. After reordering the fields, the new size of the struct is checked to see if there is any improvement. If yes, the struct would have its fields reordered, otherwise the struct would remain with the same ordering of fields. The IR needs to be further modified to ensure that the program's functionality is not affected from the size changes of the structs. This is done by updating the memory allocation sizes given to any dynamic AoS pointers, and also updating GEP instructions that access the modified struct fields.

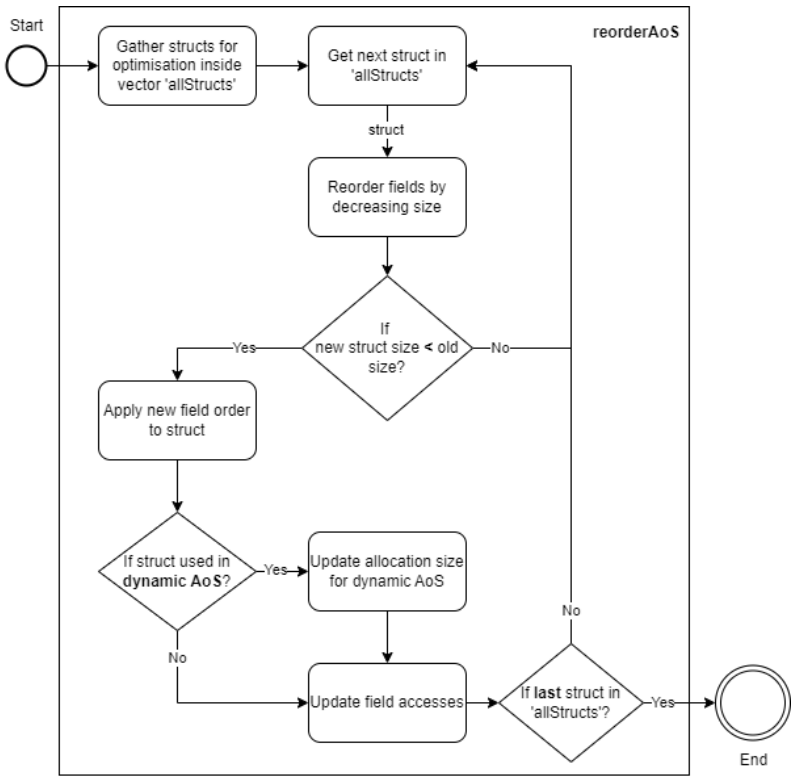


Figure 27: Top-view design of the structure field reordering pass

There are three processes that are undergone when reordering the struct fields. Firstly, information about each struct field is collected and stored in a vector called `elems`, as shown in Listing 23. The sizes of each field of a struct (in bits) is determined using the function `getTypeSizeInBits()` that is available from the `DataLayout` class [35]. Initially, the value of the new index for each field is set to 0, and this will be updated when the fields are reordered. Secondly, the fields inside the `elem` vector are sorted by decreasing size and stored in a new vector called `sortedElems`, which shares the same container structure as the `elems` vector. In this stage, the 'new index' values are updated to store the new index position of the field within the struct. This index becomes useful later in order to update the GEP instructions.

Listing 23 C++ code for the vectors used to store the unsorted and sorted struct fields

```
/* Following information about each field in a struct is stored in this vector:
1) Type
2) original index
3) new index
4) size in bytes
5) boolean to indicate if it is a bitfield
*/
vector<tuple<Type*,int,int,int,bool>> elems; // stores fields in original order
vector<tuple<Type*,int,int,int,bool>> sortedElems; // intermediary vector
vector<tuple<Type*,int,int,int,bool>> newSortedElems; // sorted struct fields
```

Simply ordering from largest size to smallest size is not the best solution, because there might be large padding spaces that are remained unused, which could be filled up by smaller struct fields found at the end. Therefore, the last stage involves finding the padding space after each struct field in the `sortedElems` vector, which is done by subtracting the field size by the word length (8 bytes), as shown in Listing 24. For any padding bytes greater than zero, the fields inside the `sortedElems` vector are iterated again to see if they can be inserted inside this padding space. The final ordering of the struct fields is stored in a vector called `newSortedElems`, shown in Listing 23.

Listing 24 C++ code for vector 'elems' that stores each struct field

```
/* calculate the number of padding bytes needed */
int size = get<3>(*it1); // get third index value from field in 'sortedElems'
int padding = 0;
if(size > 8)
  padding = 8 - (size % 8);
else
  padding = 8 - size;
```

The final order of the struct fields are applied to a temporary struct variable, and the new struct size can be determined from this struct using a function called `getTypeAllocSize()` from the `DataLayout` class, which returns the size of the struct including the padding bytes [35]. Given that this new struct size is smaller than the original struct size, the new order of fields are officially applied to the original struct.

If a struct size has been reduced, the total size of an AoS would also be reduced. This means that size of dynamic AoS pointers must be reduced, which has memory allocated using `malloc()` or `calloc()` functions. The original size of these pointers can actually remain unchanged as it is enough to store the struct elements, however this results in some of the allocated memory space being unused. This is not ideal for memory-constrained systems or programs that required a lot of memory, therefore it was decided that the compiler pass should adjust the size of memory allocated to these dynamic AoS data structures. This is done by obtaining the original `malloc()` or `calloc()` function call. A `malloc()` function call is replaced with a new `malloc()` function call, that allocates memory equivalent to the new struct size in bytes. A `calloc()` function call only requires the second operand to be updated to the new struct size.

For all AoS data structures, all GEP instructions must be updated so that they access the correct indices after the fields have been reordered. As shown in Lines 12-13 of Listing 18, a struct field access from an AoS is represented using two consecutive GEP instructions: the first GEP instruction accesses a struct element from the AoS and the second GEP instruction accesses a particular field from the returned struct

element. For this optimisation, the second GEP instruction must be updated, otherwise the struct fields would not be accessed correctly which can break the program's functionality and outputs. When a GEP instruction returns the optimised struct for the function `getSourceElementType()`, this means that the fields of the optimised struct are being accessed i.e. this is the aforementioned second GEP instruction. The last operand of this instruction signifies the index of the struct field being accessed. Using the index, the correct field is found in the vector `newSortedElems`, and from this vector element, the new index value is retrieved and applied to the last operand of the GEP instruction.

This whole process of reordering struct fields and updating field accesses is performed for all structs stored in the `allStructs` vector. Once the end of this vector is reached, the optimisation is complete.

3.3 Implementation of Structure Peeling

The 'peelAoS' compiler pass performs the structure peeling optimisation on structs belonging to global AoS and local, static AoS data structures **only**. Additionally, structs containing pointer fields or have been used in function arguments **cannot** be peeled. Therefore at the start of this compiler pass, all of the valid structs are obtained and stored in the `allStructs` vector, by checking the details about each AoS in the `confirmed` vector that was created in the 'detectAoS' pass.

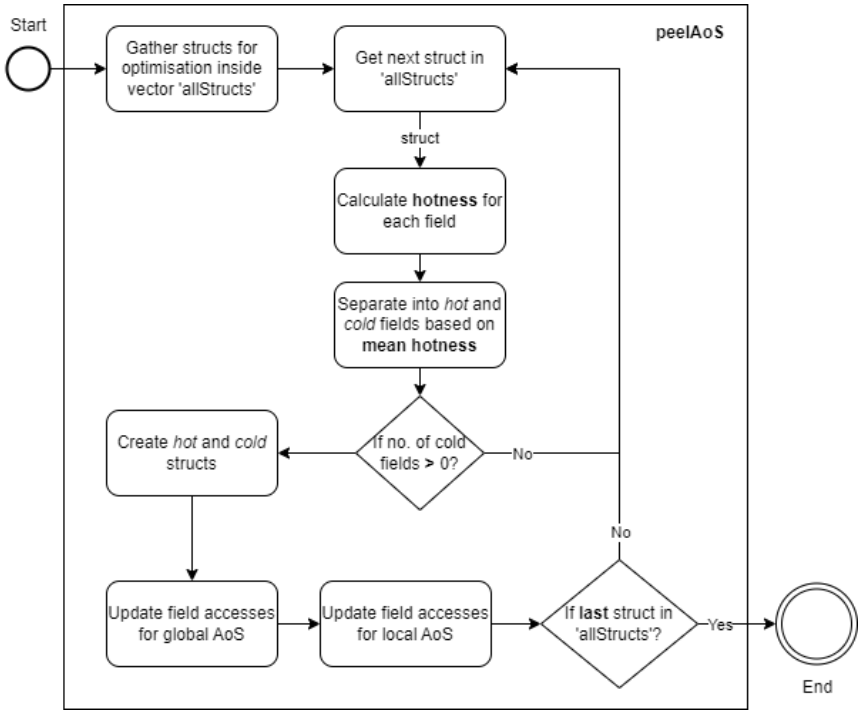


Figure 28: Top-view design of the structure peeling optimisation pass

Figure 28 shows the overall structure of the 'peelAoS' compiler pass. Optimisation begins by calculating the *hotness* value for each struct field, which defines the frequency of accesses within the program. This is done by iterating through all of the IR, looking for GEP instructions that access the struct fields. The last operand of a GEP instruction specifies what field is being accessed. From this, a counter value is updated to keep track of the number of times the field was used. The counter values for each struct field is stored in the vector `elems` as shown in Listing 25. This vector will have a size equivalent to the number of struct fields, so each vector element represents a struct field of a struct currently being optimised, represented by the variable `currStruct`.

Listing 25 C++ code for initialisation of vector 'elems' that stores the *hotness* value for each struct field

```
vector<float> elems;
for(int j = 0; j < currStruct->elements().size(); j++) //counts set to zero
    elems.push_back(0.0);
```

After calculating the hotness values for each struct field, the mean value is calculated. The mean is used to determine which field belongs to the 'hot' and 'cold' structs. Hotness values greater than or equal to this mean are placed in the hot struct, whilst values less than the mean are placed in the cold struct. As shown in Listing 26, the separated fields are stored in the `hotFields` and `coldFields` vectors. These are later used in lines 8 and 11-12 to create the 'hot' and 'cold' structs respectively. Line 8 shows that the fields of the original struct is only replaced in order to create the 'hot' struct, whilst Lines 11-12 shows that a new struct has to be created for the 'cold' struct. Whilst the fields are being separated, the original and new indices of each 'hot' and 'cold' field are stored in the vectors `hotIndices` and `coldIndices` in that order. These pair values for each field would be used to correctly update the field accesses of GEP instructions.

Listing 26 C++ code of vectors used for separating fields into 'hot' and 'cold' structs

```
1  vector<Type*> hotFields;
2  vector<pair<int,int>> hotIndices; // stores orig and new indices
3
4  vector<Type*> coldFields;
5  vector<pair<int,int>> coldIndices; // stores orig and new indices
6
7  // Assigning hot fields to original struct
8  allStructs.at(i)->setBody(newHotFields);
9
10 // Creating cold struct
11 StructType* coldStruct = StructType::create(peel_Context, coldName);
12 coldStruct->setBody(coldFields); // Assigning cold fields
```

Since a new 'cold' struct is created for each AoS element, these structs need to be stored in a new AoS data structure. This array will be referred to as the `coldArray`. If a global AoS is being optimised, the `coldArray` would be implemented as a global variable, as shown in line 3 of Listing 27. Since this is a global variable, it only needs one declaration in order to be accessed anywhere in the program. If a local, static AoS is being peeled, the corresponding `coldArray` would be implemented as a local declaration, as shown in in line 9 of Listing 27. This `coldArray` needs to be declared within the function where the original AoS has been declared, which is easily obtained from the `confirmed` vector. It is declared here because all of the field accesses would be present inside the originating function, since this AoS will not be used as function arguments and accessed in other functions.

Listing 27 C++ code for creation of cold AoS as a global declaration

```
1 /* creating new global AoS that has same attributes and size as original AoS, but
  ↪  it now stores the cold struct as elements */
2 Type* newTy = ArrayType::get(coldStruct,globalSize);
3 GlobalVariable* coldArray = new GlobalVariable(M, newTy, false, linkageType,
  ↪  Constant::getNullValue(newTy));
4 coldArray->setName(globalName); // set name
5 coldArray->setAlignment(globalAlign); // set alignment
6 coldArray->setDSOLocal(isDSOLocal); // set this attribute to be same as original
7
8 /* Creating new local AoS to store cold structs */
9 coldArray = new
  ↪  AllocaInst(ArrayType::get(coldStruct,numElem),0,size,alignment,name,aosLocation);
```

Lastly, this compiler pass is responsible for correctly updating the field accesses of the new 'hot' and 'cold' structs. As described previously, two GEP instructions are present consecutively to access the array and the struct in that order. If a 'hot' field is being accessed, the last operand of the second GEP instructions only requires change; this is updated with the new index of the 'hot' field stored in the vector `hotIndices`. If a 'cold' field is being accessed, both GEP instructions must be modified. The first GEP instruction must access the `coldArray` AoS instead of the original AoS, and also return a 'cold' struct instead of the original struct. These modifications are done using the function calls `setSourceElementType(coldArray)` and `setResultElementType(coldStruct)` respectively. The second GEP instruction must access the cold struct returned by the first GEP instruction, which is set by the function calls`setSourceElementType(coldStruct)` and `setOperand(0,coldArray)` [24]. Additionally, the last operand of the second GEP instruction must be changed so that it accesses the correct field from the 'cold' struct. This index is retrieved from the previously-defined vector `coldIndices`.

3.4 Implementation of Structure Splitting

The 'splitAoS' compiler pass performs the structure splitting optimisations on AoS structs. In theory, this optimisation can be applied to all structs legally i.e. without breaking the program's functionality. However, due to the presence of the 'peelAoS' pass, this optimisation is only targeted towards dynamic AoS data structures and AoS data structures used as function arguments. Figure 29 shows the overall structure of the 'splitAoS' optimisation pass.

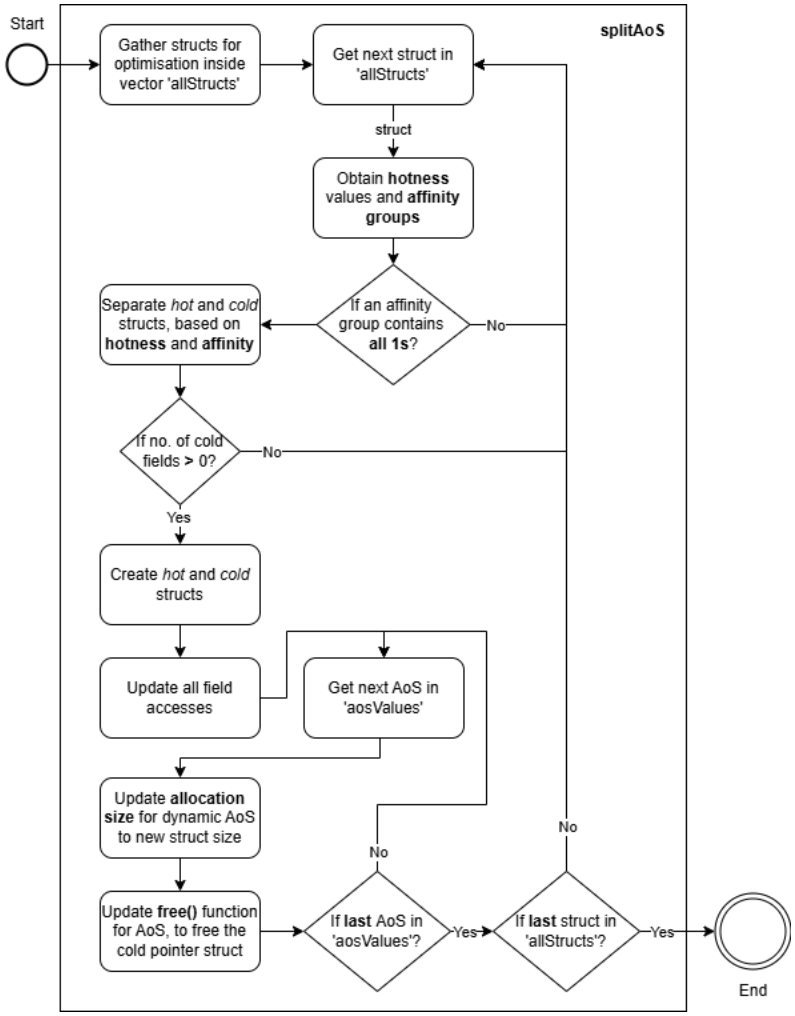


Figure 29: Top-view design of the structure splitting optimisation pass

Unlike structure peeling, the fields in each struct are separated based on two metrics: *hotness* and *affinity*. Hotness is the same as what was used in structure peeling - it measures the frequency of uses of each field. Affinity measures how close each field is accessed in time [22]. In terms of the IR, this means checking what fields of the struct are accessed together in the same BasicBlock of instructions, where each BasicBlock could represent a function body, a `for` loop body or a `while` loop body. Every in-

struction in the IR is analysed and the hotness and affinity group values are computed and stored in the containers `elems` and `fieldAccessPattern`, as shown in Listing 29. The `elems` vector is identical to the one seen in the 'peelAoS' pass. The new set `fieldAccessPattern` stores a vector of integers that represents the affinity groups, which each vector size equivalent to the total number of fields in the struct and the integers of each vector initially set to 0. For example, for a struct with four fields, each integer vector is initialised to 0000. Within a BasicBlock, an integer of a vector is set to 1 if the corresponding struct field (represented by a particular index value of the vector) is being accessed. For example, if the first field of the struct is accessed, the first integer of the vector is set to 1. This will result in a string of 0s and 1s that is later used with the hotness values to organise the 'hot' and 'cold' fields.

Listing 28 C++ code for vectors used in structure splitting

```
set<vector<int>> fieldAccessPattern; // each element stores groups of fields that
↪  are accessed together
vector<float> elems; // stores counts of each field access
```

With the hotness values and affinity groups computed, the 'hot' fields are first organised by comparing each hotness value to the mean hotness. Fields that are not organised into the 'hot' fields are automatically assumed to be 'cold' fields but this is not final, since some of these fields can be added to the 'hot' struct depending on the affinity groups. After doing this, each vector (affinity group) inside the set `fieldAccessPattern` is analysed. If there is affinity group that has all fields accessed (all integers set to 1), this means that there is good affinity within the struct as it is being used completely. Therefore, there is no need to split the struct. Performing this optimisation on a struct with good affinity can ruin the runtime performance, since more clock cycles are required to fetch both the 'hot' and 'cold' struct in order to access all fields. Given that there is no such affinity group, the compiler pass proceeds to find any struct fields that could be added to the 'hot' struct. This is done by checking whether a 'cold' struct field is accessed in the same BasicBlock as another 'hot' field access. If so, this field should not be in a 'cold' struct and should be placed inside the 'hot' struct instead. The aim of this is to ensure that the 'hot' struct is used as efficiently as possible, by considering the access patterns of the fields as well as the frequency of the accesses.

Once all the affinity groups have been analysed, the 'hot' and 'cold' structs are created, given that there exists some 'cold' fields. Instead of just creating a new structure for the 'cold' struct, a pointer field of the 'cold' struct type is created and inserted to the 'hot' fields shown in Listing 29. Unlike with the 'peelAoS' pass, there is no need for the declaration of another AoS to store the cold structs; the 'cold'

struct for each AoS element is stored within the corresponding ‘hot’ struct.

Listing 29 C++ code for creating the ‘cold’ struct pointer and adding it to the ‘hot’ fields

```
PointerType* coldStructPtr = PointerType::get(coldStruct, 0); // Initialise the
↪  cold struct pointer

hotFields.push_back(coldStructPtr); // add the cold struct ptr to the 'hot'
↪  fields
```

As seen with the other optimisation passes, the struct field accesses of every GEP instruction must be updated. For ‘hot’ field access, this requires a simple change to the last index operand of the GEP instruction, to reflect the new index of the field. For ‘cold’ structs however, it requires more complex changes, highlighted in Figure 30. A new GEP instruction (coldGEP) must be created that access the ‘cold’ struct from the pointer field. If this pointer has no memory allocated to it, it is done so using a `malloc()` function call. This allocated memory is stored into the pointer using a `store` instruction and a `load` instruction of this pointer is created so that the coldGEP can be accessed easily. For the current GEP instruction representing the field access, it is updated so that it access the ‘coldStruct’ that is produced by the coldGEP. This is done using the function calls `setSourceElementType(coldStruct)` and `setOperand(0,coldGEP)`.

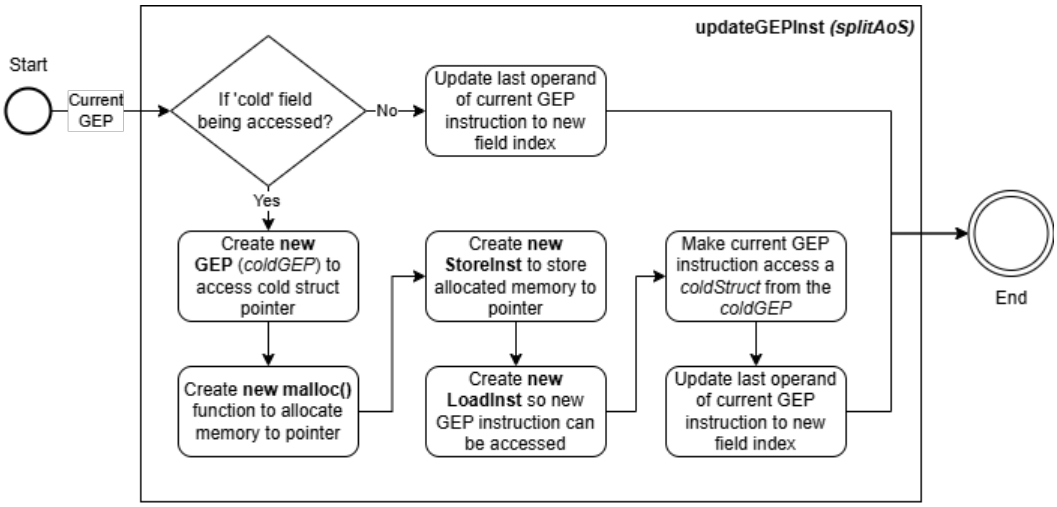


Figure 30: Steps required to update field accesses within ’splitAoS’ pass

After updating the field access, every optimised AoS must undergo two changes: updates to the allocation size and de-allocation of the ‘cold’ struct pointer. The first step involves changing the allocation sizes given to dynamic AoS pointers, in order to reflect the reduced size of the structs. This is performed only on dynamic AoS pointer to make sure that memory is not wasted. Secondly, a new `free()` function

needs to be implemented and inserted at the end of the program in order to free the memory allocated to the 'cold' struct. This procedure is done as part of good C programming practice and to avoid memory leaks. This `free()` function is implemented using a `while` loop in the IR, which iterates through each AoS struct elements and de-allocates the 'cold' struct pointer.

3.5 Detection of SoA

The 'detectSoA' pass is almost identical to the 'detectAoS' code in terms of the overall code structure. As shown in Figure 31, the main difference between this pass and the 'detectAoS' pass is the first 'CollectSoA' phase, which detects SoA data structures as well as initialises new vector containers to store these SoA data structures. The 'DetectionOfSoA' phase is identical in operation to the 'DetectionOfAoS' phase of the 'detectAoS' pass shown in Figure 22, with a few differences applied to the variable and data structure names, so that the detection of SoA data structures can be created as an individual compiler pass.

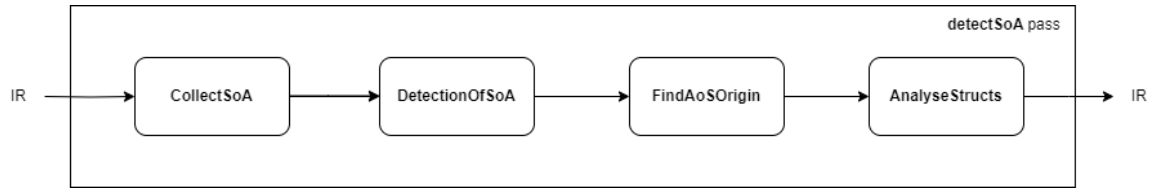


Figure 31: General structure of 'detectSoA' compiler pass

Before the `main()` function is searched for SoA data structures in the same way as it is done in the 'detectAoS' pass, all of the structs used in the program are checked. This is obtained using the function `getIdentifiedStructTypes()` available from the `Module` class [27]. For each struct, all of its fields are analysed to see if it is an `ArrayType`. For a correct definition of an SoA in this project, all struct fields must be an array. Therefore, all fields must be an `ArrayType` in order to detect an SoA data structure.

Once an SoA has been found, it is stored in a new map container called 'toFind'. A map is used here to prevent storing duplicate SoA values, and it also allow for easier searching using a `<key,value>` pair, which is represented using `<Type*,int>` in the compiler . The key component is the actual SoA represented as a pointer to a `Type` instance, which is the parent class of `StructType` so both classes are equivalent. The value component of the pair represents the size of each SoA, which is obtained using the function `getTypeAllocSize()` from the `DataLayout` class [35].

Listing 30 Declaration of the map 'toFind' in the detectAoS.h header file

```
// used by 'detectSoA' to store all SoAs and its sizes
map<Type*,int> toFind;
```

Once all SoA data structures have been detected, the last two phases in Figure 31 perform the same operations as implemented in the 'detectAoS' pass. These phases correctly update the information of each SoA in the 'confirmedSoA' vector, store each SoA size in the 'origStructSizes' vector and detects any pointer fields with each SoA. Since there are no optimisations for SoA data structures currently available, this means that these vectors remain unused for SoA data structures, therefore these processes remain redundant. However when optimisations are made available to SoA in future work, these vectors are readily available for use and can speed up implementation of these new optimisation passes.

3.6 Detection of AoSoA

The detection of AoSoA data structures is performed by running the 'detectSoA' pass first then performing the 'detectAoS' pass straight after. The reason for this is because the 'detectSoA' pass stores all of the SoA data structures in the map 'toFind', which is then used by the 'detectAoS' to check whether an array contains SoA elements. It was not decided to implement this detection as a separate compiler pass, since it involves unnecessary duplication of code. Instead, a small amount of code can be added to the 'detectAoS' pass in order to complete this detection.

Whenever a GEP instruction is accessing a struct from an array, the function `getSourceElementType()` from the `GetElementPtrInst` class should return a `StructType`. If this is true, the `StructType` is searched for inside the 'toFind' map. If found, this means that an SoA is being accessed from an array, therefore an AoSoA data structure is present. As shown in Line 6 of Listing 32, a boolean flag is set true which would be stored with the AoS in the 'confirmed' vector, to indicate that it is an AoSoA.

Listing 31 Process of detecting an AoSoA within the 'detectAoS' pass

```
1 if(auto *ST = dyn_cast<StructType>(gep->getResultElementType()))
2 {
3   gepStruct = ST;
4   // //check if struct is in 'toFind' and set bool to true
5   if(toFind.find(gepStruct) != toFind.end())
6     isAoSoA = true;
7 }
```

4 Testing

For testing and collecting results, each optimisation pass will be applied to an unoptimised program, and its execution time and memory consumption will be measured. Firstly, the unoptimised program will be run before applying the optimisations and the results are collected. These unoptimised results are then compared with results of the optimised program, which have the optimisations applied.

The optimisation passes were tested on the DCS Batch Compute system, instead of using a personal computer or the DCS lab computers. This is because the DCS Batch system only runs the bare necessities of a Linux operation system, so there would be no background process running. So all of the CPU time and memory can be allocated to running the unoptimised and optimised programs, meaning that there are no disruptions to the accuracy of the results. Listing 32 shows the information about the CPU architecture used for the DCS Batch system, which is obtained by running the `lscpu` command on the terminals.

Listing 32 Information about the CPU used for the DCS Batch Compute system

```
===== ENVIRONMENT =====
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              40
On-line CPU(s) list: 0-39
Thread(s) per core:  2
Core(s) per socket:  10
Socket(s):           2
NUMA node(s):        2
Vendor ID:           GenuineIntel
CPU family:          6
Model:               63
Model name:          Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz
Stepping:            2
CPU MHz:             3300.000
CPU max MHz:         3300.0000
CPU min MHz:         1200.0000
BogoMIPS:            5199.99
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            25600K
NUMA node0 CPU(s):   0-9,20-29
NUMA node1 CPU(s):   10-19,30-39
```

The CPU has multiple cores / threads available, however the test programs are only single-threaded. Therefore for testing, only one core / thread is required and used to run each unoptimised and optimised program. Multiple cores / threads are not used for testing since this does not change the runtimes or memory usages of the programs that are being run.

Listing 33 Example command for running an optimised program using an AoS size 100000 on a DCS Batch Compute node

```
./remoterun.sh ./optimised 100000
```

The bash script `remoterun.sh` is used to run the unoptimised and optimised programs independently, as the script is responsible for assigning a compute node to an executable. As shown in Listing 34, certain parameters are given as Slurm commands, which gives each job an ID, a single CPU, a maximum memory limit of 16 GB and a maximum time limit of 24 hours for execution [36].

Listing 34 Slurm commands provided for each job running on the DCS Batch Compute system.

```
#SBATCH --job-name=$( whoami )-$1
#SBATCH --cpus-per-task=1
#SBATCH --mem=16G
#SBATCH --time=1-00:00:00
```

For each executable running on a node, the `time` command is used with the verbose option `-v` in order to print out the execution times of the executable onto a text file [37]. An additional command `seff` is provided by Slurm to print out the memory utilisation of each executable after completion [38]. The unoptimised and optimised programs were run on increasing AoS sizes, ranging from 1,000 to 1,000,000 struct elements. Five runs were made for each AoS size, with the runtimes and memory consumption results collected for each run. This is done to remove any outliers so an accurate mean value can be computed for each AoS size. The mean values are computed by a MATLAB script, which is also used to produce the bar graph plots that are shown in the following sections.

4.1 Structure Field Re-ordering

The program `unordered.c`, used to test the structure field reordering optimisation, consists of two AoS data structures: a globally-declared static AoS called `arrayOne` and a locally-declared dynamic AoS called `arrayTwoOld`, which both use the same struct `nodeOneOld`. Both AoS data structures are populated and are used by the functions `calculationsOne()` and `calculationsTwo()`. These functions include a series of arithmetic calculations that are run 100,000 times for each AoS element using a

`for` loop, which is done in order to increase the number of computations and memory accesses. After this functions are called within the `main()` function, all fields of a particular struct element from the `arrayTwoOld` AoS is printed out, which is later used to check with the outputs of the optimised program to confirm that the optimisation does not break the program.

```
struct nodeOneOld
{
    char e;
    double g;
    int c;
    long long int d : 48;
    double f[2];
    double h;
    double i;
    int a;
    double b;
    float j;
};
```

Listing 35: Unoptimised struct (unordered fields)

```
struct nodeOneOld
{
    double f[2];
    double b;
    double g;
    double h;
    double i;
    long long int d : 48;
    char e;
    int a;
    int c;
    float j;
};
```

Listing 36: Optimised struct (ordered fields)

Figure 32: Comparison between the unordered and ordered struct in C

The struct used by both AoS data structures, `struct nodeOneOld`, has an original size of 88 bytes. Since a single clock cycle on a 64-bit architecture accesses 8 bytes, it requires a total of 11 clock cycles to fully retrieve the struct. After reordering the fields from the largest size to the smallest size, as shown in Listing 36, the re-ordered struct now consumes 72 bytes in memory, which requires 9 clock cycles to fully fetch. This means that after applying this optimisation, the struct size has been reduced by 16 bytes and requires two less clock cycles to process. This reduction in struct size has effects to the execution times and total memory consumption, as shown in Tables 5 and 6.

Table 5: Structure Field Reordering results table - Execution time vs. AoS size

AoS size	Unoptimised time	Optimised time	Time difference	Speedup
1000	2.44 seconds	2.21 seconds	0.23 seconds	**9.55%**
5000	12.23 seconds	11.01 seconds	1.23 seconds	**10.02%**
10000	24.07 seconds	21.50 seconds	2.57 seconds	**10.68%**
50000	131.31 seconds	111.90 seconds	19.41 seconds	**14.78%**
100000	305.51 seconds	277.85 seconds	27.66 seconds	**9.05%**
250000	831.05 seconds	704.27 seconds	126.79 seconds	**15.26%**
500000	1651.14 seconds	1381.07 seconds	270.06 seconds	**16.36%**
750000	2461.94 seconds	2100.47 seconds	361.47 seconds	**14.68%**
1000000	3340.37 seconds	2785.82 seconds	554.55 seconds	**16.60%**

The results for the execution times of the optimised program in Table 5 shows that the *speedup*, which is the time difference over the unoptimised time, is constant at around **10 - 15%** for all array sizes. The reason for this is because the structure size is reduced by a fixed amount of 16 bytes, which means that it always requires less than two clock cycles to access the optimised struct for all AoS sizes. With a smaller struct size, each AoS element is smaller in size so more of these array elements can be stored within the available cache memory. This means that the program is less reliant on slower main memory accesses, since it is likely that the required struct is found inside much faster cache memory. Despite the speedup being constant, programs with larger AoS sizes experience a much larger time reduction as shown by the orange bars of Figure 33. This indicates that structure field reordering is not very effective on smaller AoS sizes, as it does not produce significant time reductions.

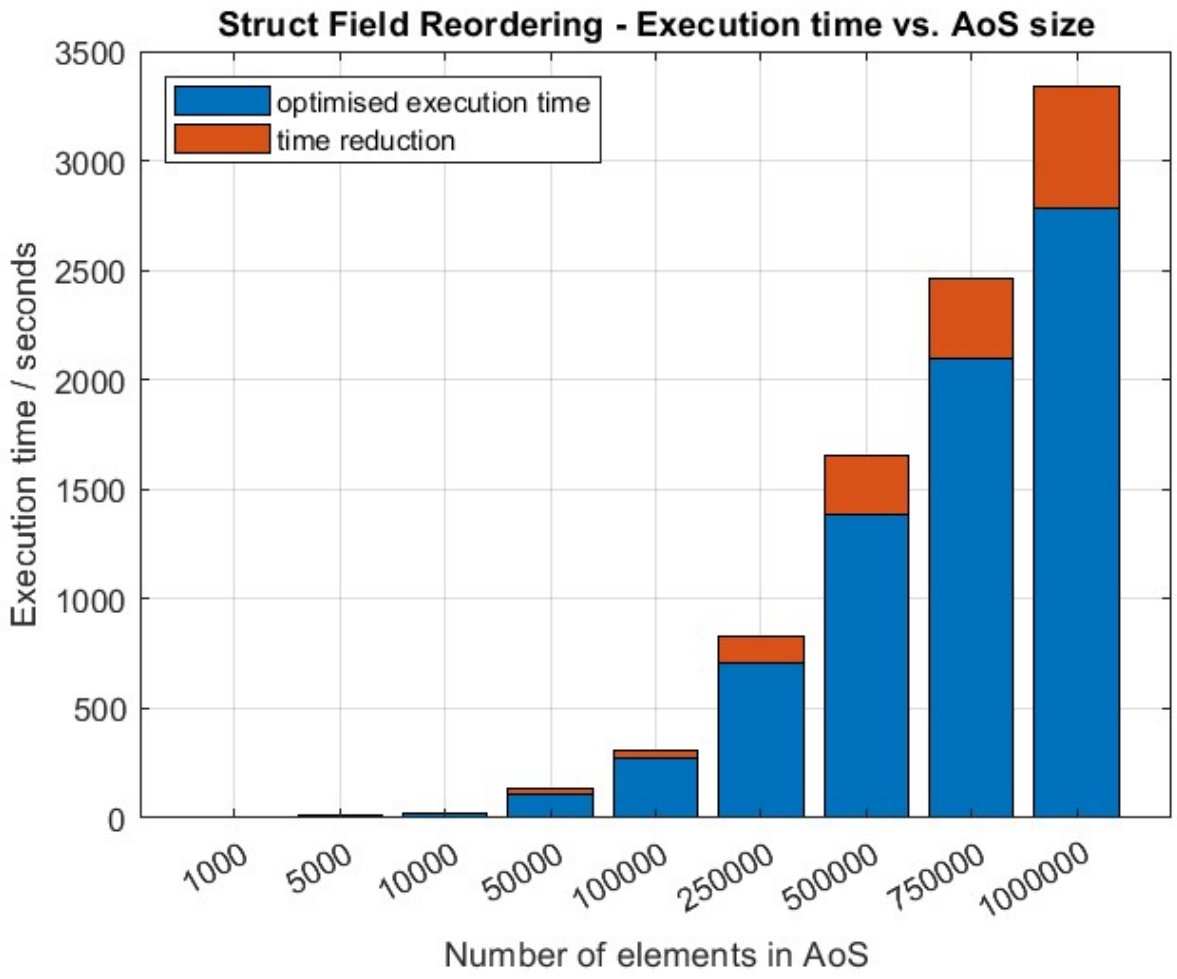


Figure 33: Structure Field Reordering bar graph - Execution time vs. AoS size

Table 6 shows the memory consumption being reduced by a constant amount between **16 - 18%** for all array sizes. The reason for this is because the structure size of each array element is reduced by a fixed

amount of 16 bytes. Figure 34 also indicates that this optimisation is favourable for larger AoS sizes, since it produces more significant memory savings compared to smaller AoS sizes.

Table 6: Structure Field Reordering results table - Memory usage vs. AoS size

AoS size	Unoptimised memory	Optimised memory	Memory difference	Saved
1000	1.62 MB	1.62 MB	0.01 MB	**0.31%**
5000	1.62 MB	1.61 MB	0.01 MB	**0.41%**
10000	1.61 MB	1.62 MB	0 MB	**0%**
50000	14.78 MB	14.71 MB	0.07 MB	**0.45%**
100000	17.94 MB	14.87 MB	3.07 MB	**17.13%**
250000	43.16 MB	35.54 MB	7.62 MB	**17.65%**
500000	85.13 MB	69.86 MB	15.27 MB	**17.94%**
750000	127.06 MB	104.68 MB	22.38 MB	**17.62%**
1000000	169.05 MB	138.49 MB	30.56 MB	**18.08%**

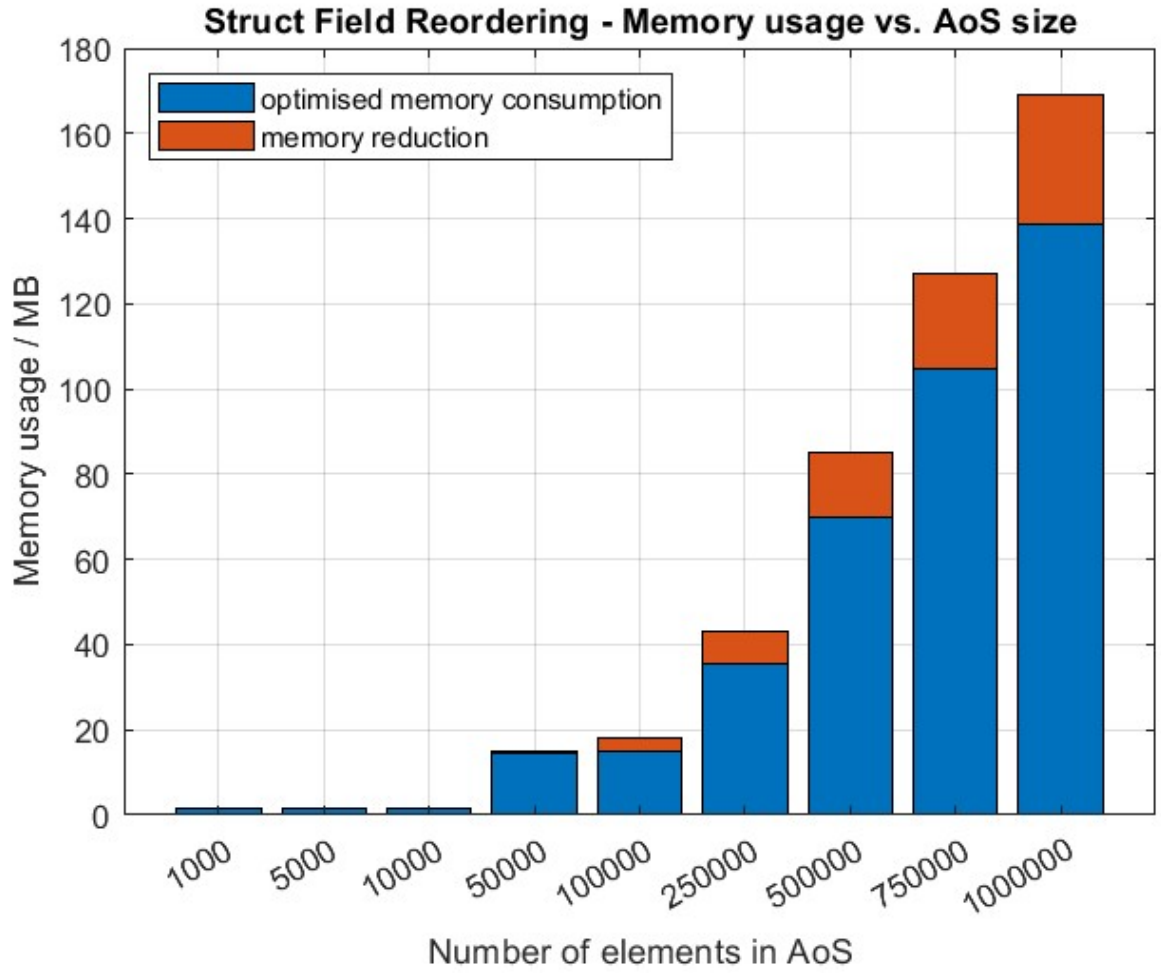


Figure 34: Structure Field Reordering bar graph - Memory usage vs. AoS size

4.2 Structure Peeling

The test program used for structure peeling, `unpeeled.c`, contains two populated global, static AoS data structures called `arrayOneOld` and `arrayTwoOld` that share the same struct. Similar to the test program used for structure field reordering, the functions `multNodeOne()` and `multArrays()` are implemented and called inside the `main()` function to perform memory-bound calculations on the struct fields of each array element. More specifically, `multNodeOne()` performs multiplication and division operations within the array elements of `arrayOneOld` only, whilst `multArrays()` performs arithmetic operations between the two AoS data structures. This is done to randomise the memory access patterns so the effects of the optimisations can be seen for various types of memory-bound operations.

Figure 35 shows the struct `nodeOneOld` before and after optimisation using structure peeling and structure field reordering. Both optimisations were used here to show the compatibility of both optimisations. Within this program, the most frequently used fields are the top three fields of the struct, which remain in the current struct to form the 'hot' struct whilst the rest of the fields are placed in the newly-created 'cold' struct. This results in the 'hot' struct having a size of 16 bytes and the 'cold' struct having a size of '576' bytes, which is smaller compared to the original size of '600' bytes.

```
struct nodeOneOld
{
    //// Hot fields - most commonly used
    ↪  fields
    int a;
    double b;
    int c;
    //// Cold fields - least used fields
    long long int d : 48;
    char e;
    double f[67];
    double g;
    double h;
    double i;
    float j;
};
```

Listing 37: Unoptimised struct (unpeeled struct)

```
struct nodeOneOld // hot struct
{
    double b;
    int a;
    int c;
};

struct nodeOneOldCold // cold struct
{
    double f[67];
    double g;
    double h;
    double i;
    long long int d : 48;
    char e;
    float j;
};
```

Listing 38: Optimised struct (peeled struct)

Figure 35: Comparison between the unpeeled and peeled struct in the C test program

Table 7: Structure Peeling results table - Execution time vs. AoS size

AoS size	Unoptimised time	Optimised time	Time difference	Speedup
1000	1.43 seconds	1.21 seconds	0.22 seconds	**15.27%**
5000	9.00 seconds	5.91 seconds	3.08 seconds	**34.27%**
10000	18.23 seconds	12.35 seconds	5.88 seconds	**32.25%**
50000	106.35 seconds	62.49 seconds	43.86 seconds	**41.23%**
100000	445.25 seconds	123.93 seconds	321.32 seconds	**72.17%**
250000	1402.93 seconds	310.30 seconds	1092.63 seconds	**77.88%**
500000	2793.93 seconds	637.74 seconds	2156.19 seconds	**77.17%**
750000	4123.40 seconds	957.36 seconds	3166.04 seconds	**77.78%**
1000000	5597.00 seconds	1246.00 seconds	4351.00 seconds	**77.74%**

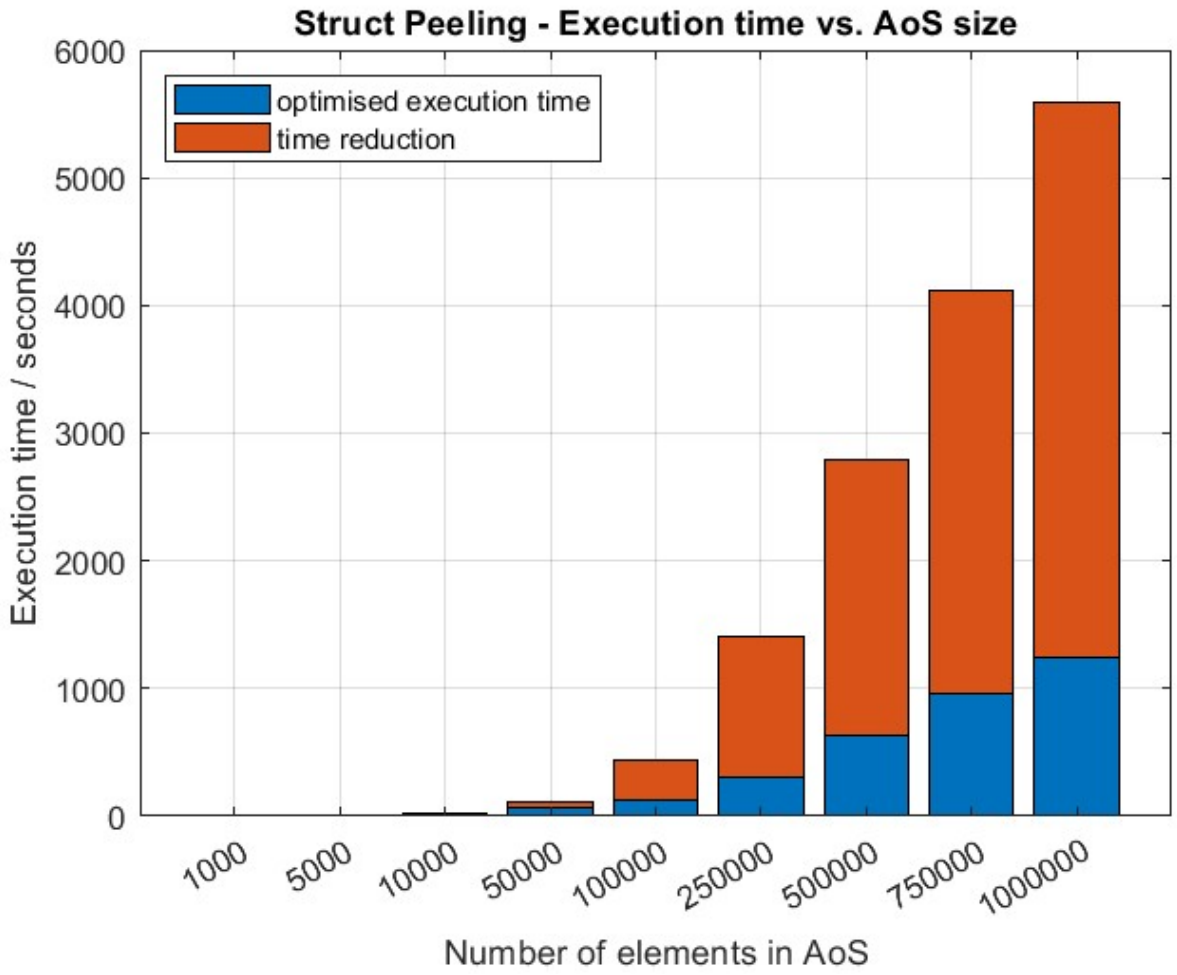


Figure 36: Structure Peeling bar graph - Execution time vs. AoS size

Table 7 shows that a significant increase in speedup is seen as the array size increases, until the size exceeds 250,000 where the speedup remains constant at around 77%. This very large speedup is resulting from the huge size reduction of the 'hot' struct, from 600 bytes to 16 bytes. Since the 'hot' fields inside

the ‘hot’ struct are predominantly used, the program no longer needs to access the 600 bytes of the original struct to use the ‘hot’ fields, which requires 75 clock cycles. By using the smaller ‘hot’ struct, it only needs 2 clock cycles to access the required 16 bytes of data; this is a massive reduction of 73 clock cycles. An additional reason for the significant speedup is that more array elements can be stored in and fetched from cache memory, due to the significantly reduced size of each struct element. This means that the program is heavily reliant on faster cache memory accesses to retrieve all or most of the array elements for computations, benefiting from spatial locality of reference. The limit of 77% when the array size increases past 250,000 elements can be explained by the restricted size of cache memory. As the array size increases, it is more likely that a required element is not stored in cache, so if it is not found in cache, slower main memory accesses are required to fetch the remaining data and update the cache with the new data. The results here store that the cache memory of the tested system can comfortably store at most 250,000 struct elements. Figure 36 also shows that the structure peeling optimisation is effective for all AoS sizes, unlike structure field reordering.

Table 8: Structure Peeling results table - Memory usage vs. AoS size

AoS size	**Unoptimised memory**	**Optimised memory**	**Memory difference**	**Saved**
1000	1.62 MB	1.62 MB	0 MB	**0%**
5000	1.61 MB	1.61 MB	0 MB	**0%**
10000	1.62 MB	1.62 MB	0 MB	**0%**
50000	58.34 MB	57.55 MB	0.79 MB	**1.35%**
100000	115.60 MB	113.97 MB	1.62 MB	**1.40%**
250000	287.20 MB	283.38 MB	3.81 MB	**1.33%**
500000	577.10 MB	565.71 MB	11.39 MB	**1.97%**
750000	861.11 MB	847.98 MB	13.13 MB	**1.53%**
1000000	1120 MB	1100 MB	20 MB	**1.79%**

Unlike seen with the execution times, the savings in memory consumption after optimisation is not very significant, as it is constant around **1 - 2%** for all AoS sizes as shown in Table 8. The small percentage in memory savings is resulting from the structure field reordering optimisation that is applied after structure peeling, which has reduced the ‘hot’ and ‘cold’ structs by 8 bytes each. However, the structure peeling optimisation does not produce reasonable memory savings on its own. This is because this optimisation only separates the original structure into two smaller structures, so the sum of the sizes of both ‘hot’ and ‘cold’ structs still sum up to the original struct size. Therefore, this optimisation is not effective at reducing memory consumption, since the optimised program uses roughly the same amount of memory as the unoptimised program as highlighted in Figure 37.

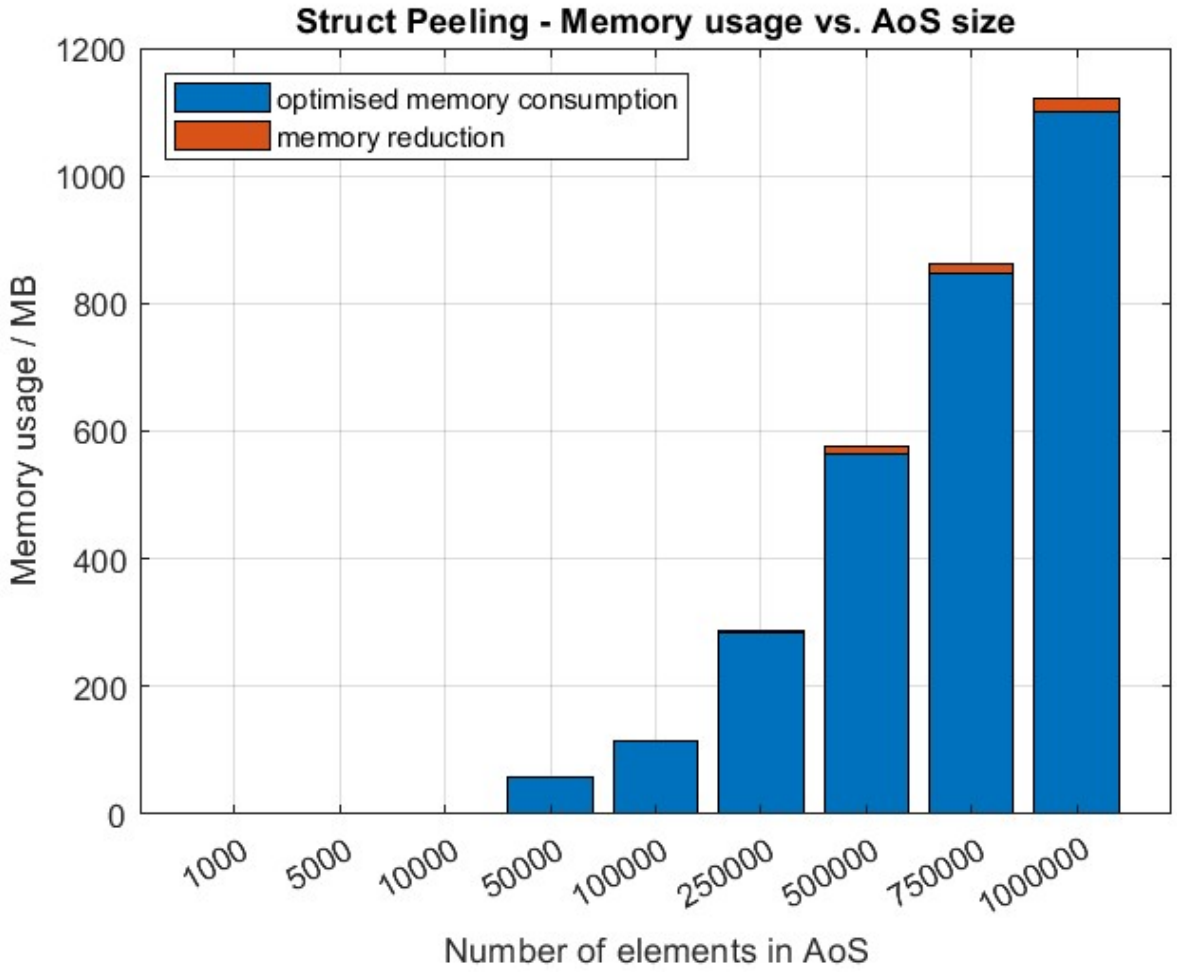


Figure 37: Structure Peeling bar graph - Memory usage vs. AoS size

4.3 Structure Splitting

The test program used for this optimisation, `unsplit.c`, is almost identical to the test program, `unpeeled.c` in terms of the arithmetic operations that are performed on the AoS data structures. The only difference is that `unsplit.c` uses two, locally-declared, dynamic AoS data structures, since structure splitting can only be applied to these AoS variants as discussed previously.

The same struct `nodeOneOld` of size 400 bytes from `unpeeled.c` is also used in this test program, which has been applied structure splitting and structure field reordering as shown in Figure 38. The 'cold' struct is the same as the struct produced from struct peeling, with a size of 576 bytes after reordering the fields. The 'hot' struct, however, is different as it includes a new pointer field to the 'cold' struct. This means that it has a size of 24 bytes, compared to the 16 bytes of the 'hot' struct produced from struct peeling, which means that a single additional clock cycle is required to access this 'hot' struct.

```
struct nodeOneOld
{
    //// Hot fields - most commonly used
    ↪  fields
    int a;
    double b;
    int c;
    //// Cold fields - least used fields
    long long int d : 48;
    char e;
    double f[67];
    double g;
    double h;
    double i;
    float j;
};
```

Listing 39: Unoptimised struct (unsplit struct)

```
struct nodeOneOld // hot struct
{
    double b;
    struct nodeOneCold* cold;
    int a;
    int c;
};

struct nodeOneOldCold // cold struct
{
    double f[67];
    double g;
    double h;
    double i;
    long long int d : 48;
    char e;
    float j;
};
```

Listing 40: Optimised struct (split struct)

Figure 38: Comparison between the unsplit and split struct in the C test program

Table 9: Structure Splitting results table - Execution time vs. AoS size

AoS size	Unoptimised time	Optimised time	Time difference	Speedup
1000	1.52 seconds	1.25 seconds	0.27 seconds	**17.63%**
5000	9.47 seconds	6.37 seconds	3.10 seconds	**32.77%**
10000	19.39 seconds	13.36 seconds	6.04 seconds	**31.13%**
50000	110.01 seconds	67.71 seconds	42.34 seconds	**38.47%**
100000	456.91 seconds	134.34 seconds	322.57 seconds	**70.60%**
250000	1476.56 seconds	347.22 seconds	1129.35 seconds	**76.48%**
500000	2943.96 seconds	709.83 seconds	2234.14 seconds	**75.89%**
750000	4387.60 seconds	1053.44 seconds	3334.16 seconds	**75.99%**
1000000	5855.20 seconds	1425.80 seconds	4429.40 seconds	**75.65%**

Despite the increased size of the 'hot' struct, the speedup still increases as the AoS size increases, and remains constant at around **76%** when the array size exceeds 250,000 elements. These results for execution time are identical to the times seen with structure peeling, and the reasons for producing these results are also the same, so it would not be repeated in this section. The difference between the speedup limit of 76% compared to 77% from structure peeling can be easily explained by the 8 bytes size increase for the 'hot' struct, as each struct element of the split AoS requires one additional clock cycle compared to a struct element of the peeled AoS. Figure 39, representing the memory consumption before and after

structure splitting, is also similar to what seen with structure peeling in Figure 36.

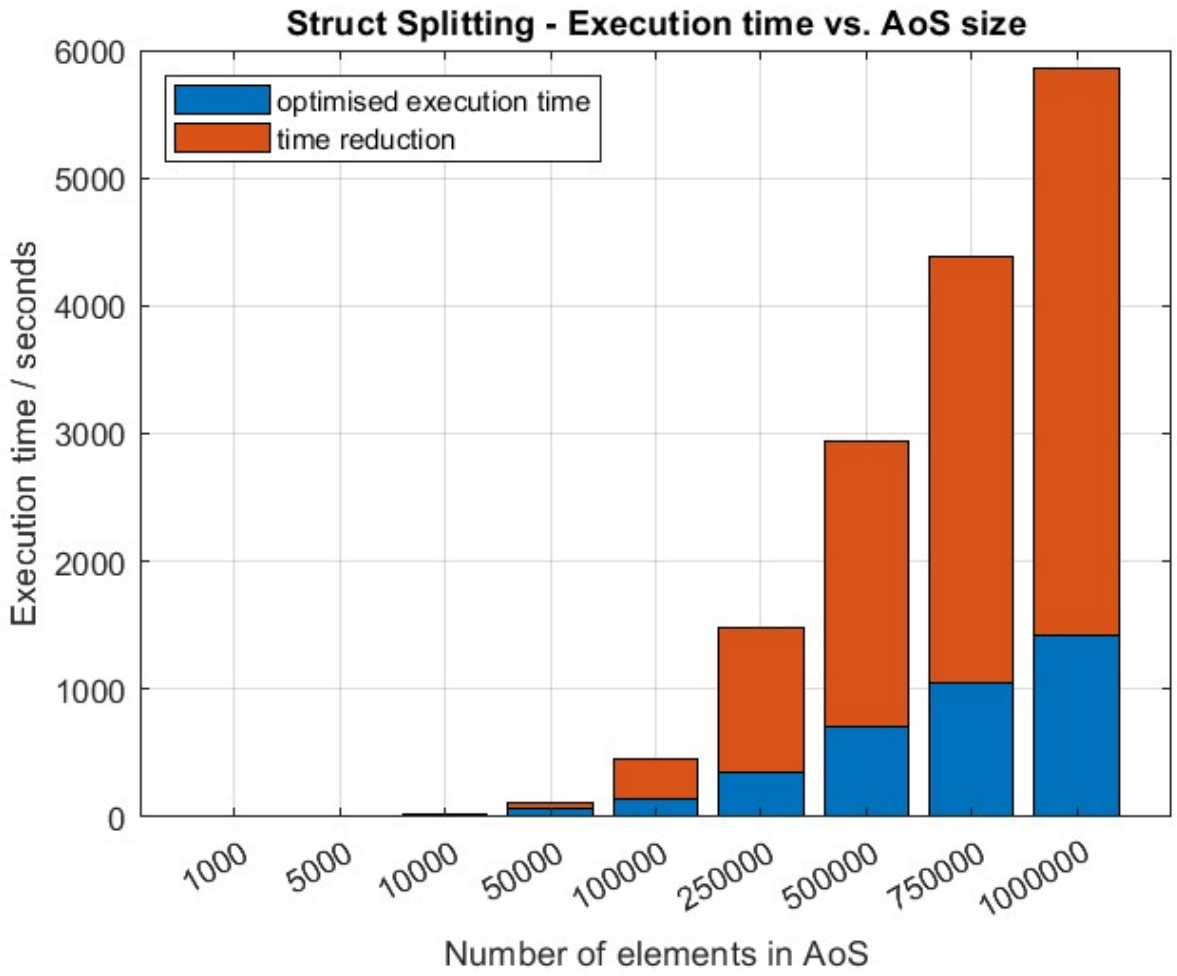


Figure 39: Structure Splitting bar graph - Execution time vs. AoS size

Despite the similarities in execution times between structure splitting and peeling, the trend in memory consumption is much different with structure splitting. Table 10 shows that the reduction in memory consumption is constant at around **46%**, which is much greater than the 1 - 2% memory reductions seen with structure peeling. The reason for this is because an new AoS data structure is not created to store the 'cold' structs for each array element, as seen with structure peeling. Instead, the 'hot' struct contains a pointer to store the 'cold' struct in heap memory. This pointer may or may not be allocated memory, which means that in the latter case, a 'cold' struct for multiple AoS elements may not be instantiated and stored in memory. In the test program, the 'cold' fields of the original struct were never used; they had a hotness value of 0. Therefore, the optimisation pass did not bother creating a 'cold' struct for each AoS element in heap memory, which explains the significant decrease in the AoS memory size.

Table 10: Structure Splitting results table - Memory usage vs. AoS size

AoS size	Unoptimised memory	Optimised memory	Memory difference	Saved
1000	1.62 MB	3.34 MB	0 MB	**0%**
5000	1.61 MB	1.62 MB	0 MB	**0%**
10000	1.62 MB	1.62 MB	0 MB	**0.21%**
50000	58.39 MB	31.55 MB	26.83 MB	**45.96%**
100000	115.61 MB	62.15 MB	53.45 MB	**46.24%**
250000	287.30 MB	153.78 MB	133.51 MB	**46.47%**
500000	573.38 MB	306.34 MB	267.03 MB	**46.57%**
750000	859.46 MB	458.96 MB	400.50 MB	**46.59%**
1000000	1120 MB	611.56 MB	508.44 MB	**45.40%**

Unlike with structure peeling, Figure 40 shows that significant memory reductions are seen for all AoS sizes, almost halving the memory consumption of the original unoptimised program indicated by the total sum of the orange and blue bars.

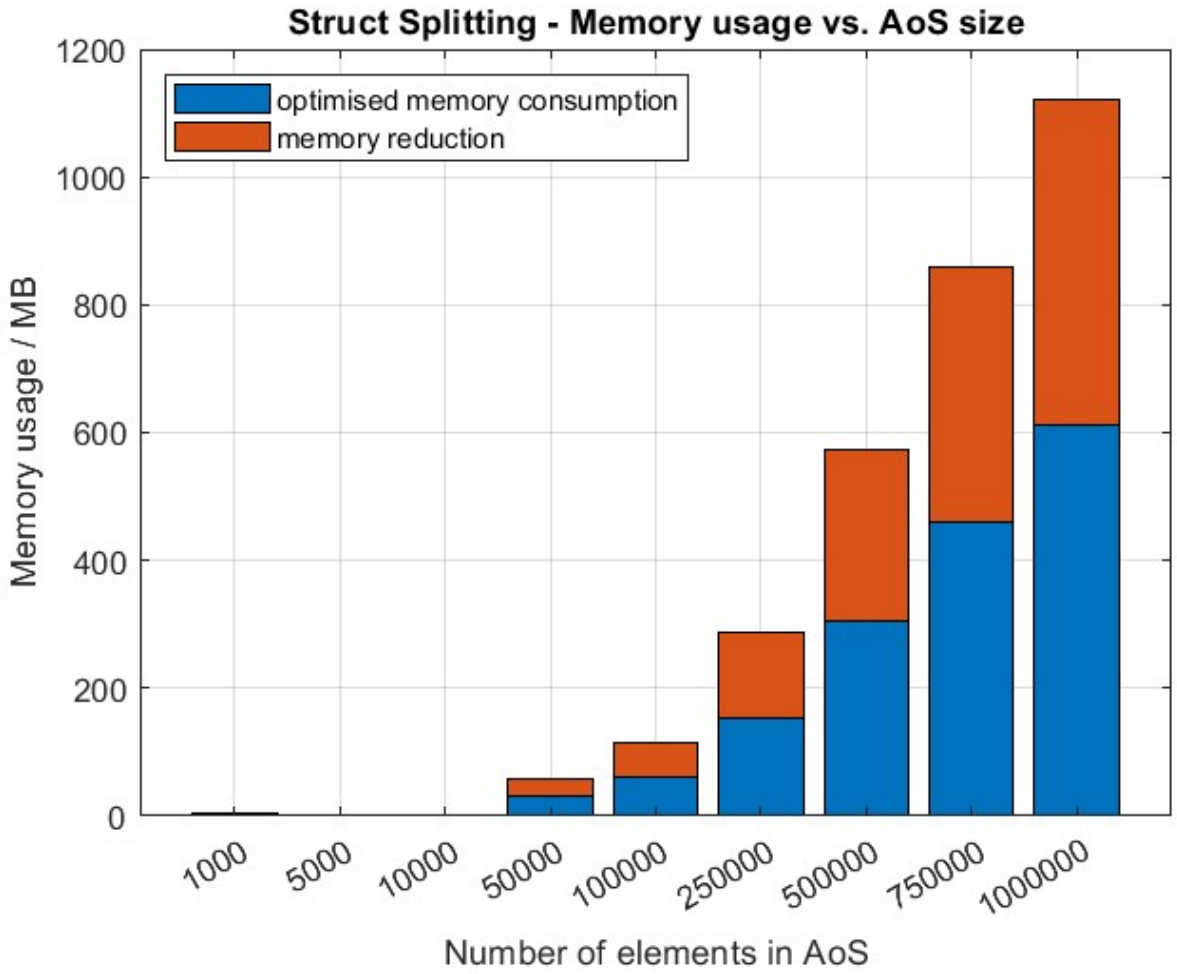


Figure 40: Structure Splitting bar graph - Memory usage vs. AoS size

5 Project Management

5.1 Methodology

For this project, an agile development methodology is adopted with aspects of plan-driven development. The agile technique of extreme programming is used, which is based around incremental development. This means that the project is broken down into several independent development cycles, also known as iterations [39]. Table 11 shows the development iterations that is devised for this project, with each iteration allocated two weeks of time for completion.

Table 11: List of development iterations for this project

Iteration number	Topic
1	Detection of static AoS
2	Detection of dynamic AoS
3	Implementation of Structure Field Re-ordering
4	Implementation of Structure Peeling
5	Implementation of Structure Splitting
6	Detection of SoA and AoSoA

By having separate, independent iterations, it makes planning more flexible to changes as the focus of each iteration can be changed without affecting other parts of the system, unlike plan-driven development where the project plan is fixed at the start and is difficult to change. Therefore, the agile nature of this project allows for swift changes to the iterations, to ensure that any project delays do not affect the overall progress of the project.

Each iteration is split into planning, implementation, and testing phases. However, the agile technique lacks the production of documentation, which is essential for creating the deliverables of this project: the progress report and the final report. So, it is decided to add an additional documentation phase to each iteration, which involves a written summary of what was achieved during the planning, implementation, and testing phases of each iteration. This documentation is very useful after project completion as it would be adapted on further during the final report when discussing the implementation of each compiler pass.

The principle of test-driven development is also utilised to make sure that each iteration is thoroughly tested before moving onto the next one [39]. These unit tests are created as demo programs during the

planning phase of each iteration. These demo programs consists of various sample data structures that are meant to be detected and/or optimised by the compiler passes. These tests ensures that the detection and optimisation pass work as intended without breaking the source program.

The agile principle of continuous integration is also used here, which made sure that each iteration can be added to the whole system easily and it can be tested with other completed iterations before becoming part of the final project [39]. This is performed by creating one top-level compiler pass, which will run each optimisation and detection passes that were implemented individually. The compiler passes are tested together using C test programs on the DCS batch compute system, as shown in Section 4, which provided an isolated environment for testing. Doing this produces reliable results on the run time and memory usage of the test programs without being affected by any other running background programs.

5.2 Tools used for project management

A GitHub repository [40] is used as version control to keep backups of the project in case of unexpected hardware failures and also provide a method for tracking progress over the project timeline, to help write the progress report and this project report. Trello, a project management tool, is also used to keep notes and checklists of what needs to be done for each iteration [41]. This helps keep the project on track by setting clear checklists that needed to be ticked off before certain deadlines. This also provides another source of project progress and event history that helped write the progress report and final report.

6 Evaluation

6.1 Reflection on Project Management

The chosen adaptation of the agile methodology has helped recover from project delays. This has proven especially useful during Term 1 of the project timeline, where iterations 1 and 2, which focused on the detection of static and dynamic AoS data structures, took longer to complete than expected. The reason for this is due to the lack of planning, where the project did not consider detecting AoS data structures within multiple levels of function calls. This error has resulted in most of Term 1 (up to Week 9) focusing on the implementation of AoS detection, as shown in Figure 42, whilst according to the original project timetable in Figure 41, the optimisations for AoS data structures should have been completed by Week 9 of Term 1.

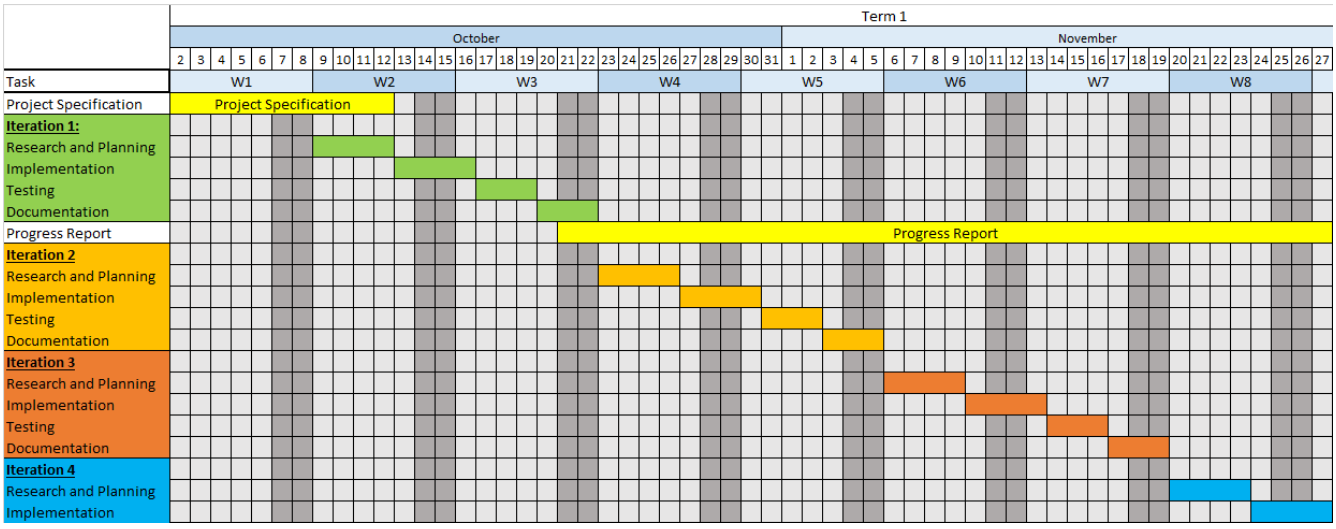


Figure 41: Caption

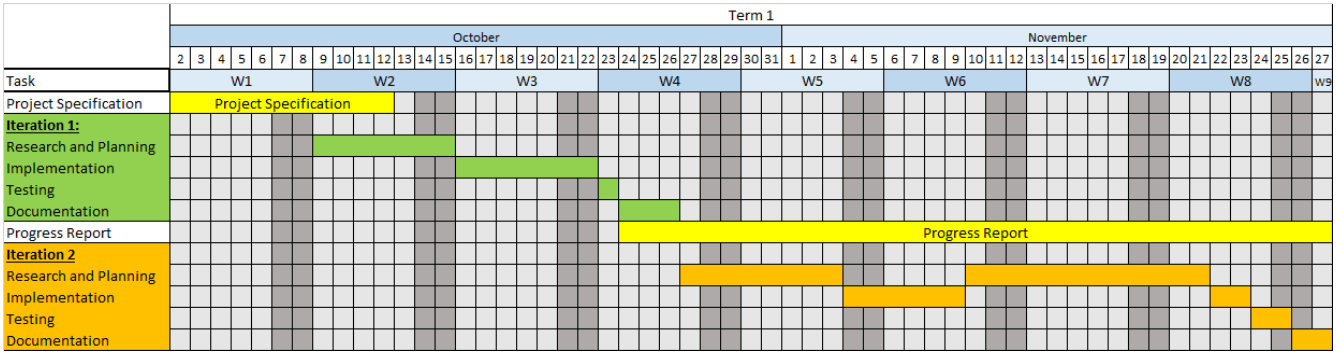


Figure 42: Actual project progress up to Week 9

Therefore, this delay means that the other iterations had to be pushed back in the Gantt charts. Luckily, this was avoided due to the time contingency plan present in the original project timetable, that was added to recover from any project delays. This was the project break period, that was allocated to the four weeks during the Christmas Break. The agile methodology allowed these incomplete iterations to be pushed into the Christmas Break so they can be completed before the start of Term 2. This change was very successful, as the incomplete iterations were finished by the start of Term 2. This also meant that the completion of iterations in Term 2 were not affected by this delay as the current project progress had caught up to the intended progress specified in the original timetable.

Furthermore, the chosen methodology allowed for easy changes to the development iterations. Two iterations were proposed for the optimisation of SoA and AoSoA data structures. Due the other commitments that caused project delays in Term 2, these iterations cannot be completed in time so they had to be removed. Even though the agile methodology helps recover from delays, it still doesn't prevent delays like with any other project management methodology, unless good planning decisions were made that considered these other university commitments during the year, such as the completion of coursework and assignments. This was difficult to do at the start of the year, since the deadlines for these assignments were not known so it couldn't be incorporated into the project timetable when writing the Specification for this project.

6.2 Reflection on Testing

Overall, the results produced from the tests show a good indication of what each optimisation is suited for. There are still improvements that could have been made to the testing process, in order to produce more reliable results.

Firstly, a variety of test programs could have been used to test each optimisation, since there are multiple ways that AoS data structures are defined and used within the programs. This was considered within the used test programs, by adding variations to the memory access patterns and using different data types for the struct fields, such as arrays. Despite this, the discoveries made on execution times and memory usages from the optimisations may not be the same for all programs. One reason why a handful of test programs were not used is because it would make testing more time-consuming within an already constrained project schedule. Additionally, one testing program is enough to verify the functionality and validity of the optimisation, which was the main focus of the testing phases.

Secondly, for each array size that was tested, more runs could have been performed. The current testing phase only performed five runs for each AoS size. Even though this helped produce accurate mean values for the execution times and memory usages, it was still not large enough to remove any outliers effectively. Several tests for certain AoS sizes had to be repeated because the outliers were not identified and removed correctly thus inaccurate values for the execution time and memory consumption were produced. Following on from this, some measurements for the memory consumption seen in Table 10 were actually outliers but these results cannot be re-computed due to lack of time within the project schedule. Even though increasing the number of runs to improve accuracy would be time-consuming for each AoS size, it would have prevented such incidents where tests have to be repeated again due to outliers, which in itself is even more time-consuming.

Lastly, the testing phase for the structure splitting only considered hotness values of the structure fields, instead of also including the affinity groups. Unlike structure peeling, the affinity of each struct fields was considered when splitting the structure. This functionality was tested using the demo programs and it proved to be beneficial. However, it would have been better to test this functionality on varying AoS sizes, as seen with the test programs in the Testing section, in order to conclude its effectiveness on different array sizes.

6.3 Reflection on Project Goals & Objectives

The project has successfully achieved the goals outlined in Section 1.2. This project has successfully implemented compiler passes that are used within the LLVM compiler infrastructure to detect and optimise data structures. Additionally, the test results prove that these optimisations successfully improve the memory-bound performance of C programs, seen by the reductions in execution times as well as memory consumption.

The original project objectives that were created in the Specification are evaluated in Figure 43, which shows the completed and incompleted objectives.

R1.1 The project **MUST** detect Array of Structures (AoS).

R1.2 The project **MUST** detect Structures of Arrays (SoA).

R1.3 The project **MUST** detect Array of Structure of Arrays (AoSoA).

R1.4 The project **SHOULD** detect Linked List data structures.

R1.5 The project **SHOULD** detect Tree data structures.

R2.1 The project **MUST** optimise Array of Structures (AoS).

R2.2 The project **MUST** optimise Structures of Arrays (SoA).

R2.3 The project **MUST** optimise Array of Structure of Arrays (AoSoA).

R2.4 The project **SHOULD** optimise Linked List data structures.

R2.5 The project **SHOULD** optimise Tree data structures.

R3.1 Each optimisation **MUST** ensure functional correctness.

R3.2 Observations **MUST** measure run-time performance for each optimisation.

R3.3 Observations **MUST** measure memory usage for each optimisation.

R3.4 The project **WON'T** optimise functions that use the detected data structures.

Figure 43: Evaluation of the project objectives at project completion

The project is successful at detecting AoS, SoA and AoSoA data structures. However, detection and optimisation passes have not been implemented for linked lists and trees, despite conducting background research on these data structures. The main reason for this is because these data structures are built up of structs that contain one or more pointer fields. As discussed already, LLVM uses opaque pointers, which means that pointers do not have types like seen with pointers in C programs [21]. This makes it extremely difficult identifying whether a pointer in a struct represents a pointer to another struct, so it is difficult to identify a node within linked list and tree data structures. This discovery of the usage of opaque pointers in LLVM was made during the implementation of structure peeling and splitting, which took place at the end of Term 1. Therefore, it was decided that it is not feasible to perform detection and optimisation of these abstract data types, due to its complexity and the constrained project schedule.

Furthermore, optimisation passes were not implemented for SoA and AoSoA data structures, despite methods such as SoA and AoSoA conversions that were discussed in this report. The constrained project schedule and the delays in the project progress has resulted in these objectives being removed for implementation, which explains why there were no iterations cited for the development of these optimisations. If there were no project delays were encountered and changes were not made to the original project schedule specified in the Specification, it would have been possible to implement at least one of these two discussed optimisations.

As detailed in the Testing section, all of objectives R3.1, R3.2, R3.3 and R3.4 have been completed, as it successfully tests the execution times and memory usages for each optimisation, as well as making sure that the programs produce the correct outputs before and after optimisation.

7 Conclusions

7.1 Summary

The project has achieved all of its goals that were outlined in Section 1.2, as it includes the implementation of five compiler passes used for detection and optimisation of specific data structures. The detection passes successfully identify various types of AoS, SoA and AoSoA data structures within the input C programs. The optimisation passes use the identified AoS data structures to apply optimisations that show good improvements in executions times and memory consumption, which concludes that memory-bound performance of the input programs have been successfully increased. Additionally, all of the implemented optimisations do not impact the accuracy and correctness of the output of the programs, thereby concluding that the optimisation passes can be safely applied to C programs.

Individual conclusions can also be made about the effectiveness of each implemented optimisation. Structure field reordering is the simplest optimisation out of three and provides minimal performance gains. Execution times were improved by a small percentage between 10 - 15%, compared to the much larger speedup percentages greater than 70% seen with structure peeling and structure splitting. In terms of reducing memory consumption, structure field reordering is only able to reduce about 16 - 18% of total memory usage. This is better than structure peeling, but not as great as structure splitting, which provided around 46% reductions in memory usage. From this, it can be concluded that structure field reordering is not effective as a standalone optimisation. Instead, structure field reordering works best when applied with other optimisations, as seen in the testing sections for structure peeling and splitting where the sizes of the 'hot' and 'cold' structs were be further reduced by this optimisation.

On the other hand, structure peeling is shown to be an effective standalone optimisation for AoS data structures, as significant speedups in execution times at around 77% can be seen for large array sizes. However, memory reductions are very insignificant at around 1 - 2%. Therefore, this optimisation is ineffective at reducing memory consumption, which means that another optimisation might need to be applied instead if memory usage is important, especially for memory-constrained systems.

Structure splitting is the most complex but most superior optimisation out the of three, as it produces significant improvements in execution times and memory consumption. Structure peeling may be slightly

better at improving execution times, since structure splitting increases the struct size with the inclusion of a pointer field. However, the exclusion of creating a new AoS data structure in structure splitting means that memory consumption can be significantly decreased, compared to structure peeling. This optimisation was very complex to implement and apply to the input program, since it performs the most changes to the IR, such as the inclusion of a pointer struct field, re-allocation of dynamic AoS pointers and the re-definition of the `free()` function to de-allocate memory.

7.2 Future work

There are several ways to extend this project as future work, in order to add improvements or new functionalities. Since SoA and AoSoA conversion optimisations were proposed but cancelled due to lack of time, it would be ideal to start implementation of these optimisations as new compiler passes. These optimisations will allow programs to reap the benefits of SoA and AoSoA data structures, in terms of improving localities between structs and between field accesses. This can be done by analysing the memory access patterns of AoS data structures and determining whether it is being used effectively i.e. if all fields of each struct in the array are accessed sequentially. If not, this data structure could be converted to an SoA or an AoSoA, depending on the analysed access pattern.

On the topic of optimisations, new compiler passes can be implemented to optimise data structures other than simple Array of Structures. In particular, optimisations could be researched and identified for SoA and AoSoA data structures, since detection passes has already been implemented for these data structures. Abstract data types (ADTs), such as linked lists or trees, can also be targeted for optimisation, since these data structures are more complex than AoS and SoA data structures thus may require memory optimisations applied to them. However as mentioned previously, the lack of pointers within the IR does not allow for easy implementation of the detection and optimisation of these data structures, unless simplifications are made. For example, the source code could include flags that will help with the detection of an ADT. By considering the optimisations of a wide range of data structures, the compiler passes within this project can have a widespread use for large amount of programs. These real-world programs may include several types of data structures instead of just simple AoS data structures that were focused on in this project.

The demo programs and test programs, used for testing the functionalities and effectiveness of each compiler pass, were manually coded by hand. These test programs were implemented to perform arbitrary

operations in order to increase the amount of memory-bound computations for testing. Therefore the performance results collected from these programs are not representative of programs that are used in the real-world, such as physics simulations which can be more complex and larger in size. To consider this, the existing compiler passes can be tested on C benchmarking software that is available online, with examples such as the STREAM benchmark [42] and the NAS benchmarks that were used by NASA [43]. These C benchmark programs may not incorporate AoS data structures which would make the optimisations in this project ineffective. Regardless, it is still worthwhile to see the real-world uses of the current compiler passes in this project and make improvements to it by identifying the types of data structures that are actually used in the real-world.

The implemented compiler passes in this project only work on C programs. As a possible improvement to this project, these compiler passes can be made to work on C++ programs as well. since C++ share some similarities with C. Furthermore, C++ is more commonly used than C, with its additional features such as object-oriented programming and data abstraction [44]. Despite the similarities between C and C++, there are also some differences that causes the LLVM IR representation for both C and C++ programs to appear widely different. Therefore, the existing compiler passes may require heavy changes which is time-consuming to perform. Disregarding any potential time constraints on future project work, integrating the current (and future) detection and optimisation passes to operate on C++ source code would be beneficial as it would improve the project's versatility for a wide range of programs.

References

[1] John L. Hennessy and David A. Patterson. “Computer Architecture: A Quantitative Approach”. In: 6th. Morgan Kaufmann Publishers Inc., 2017, 5, 49, 78, 79, 80, 81, G–25.

[2] Daniel Etiemble. “45-year CPU evolution: one law and two equations”. In: *Second Workshop on Pioneering Processor Paradigms* (2018), p. 1.

[3] *GCC: Options That Control Optimization*. `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`, last accessed on 30/3/2024.

[4] *LLVM’s Analysis and Transform Passes*. `https://llvm.org/docs/Passes.html`, last accessed on 30/3/2024.

[5] Dai Clegg and Richard Barker. *Case Method Fast-Track: A Rad Approach*. 1994.

[6] Alfred V. Aho et al. “Compilers: principles, techniques, and tools.” In: 2nd. Pearson Education, 2013, pp. 1, 358.

[7] Keith D. Cooper and Linda Torczon. “Engineering a Compiler”. In: 2nd. Morgan Kaufmann Publishers Inc., 2012, pp. 6–8.

[8] *The LLVM Compiler Infrastructure*. `https://llvm.org/`, last accessed on 5/4/2024.

[9] *Clang: a C language family frontend for LLVM*. `https://clang.llvm.org/`, last accessed on 5/4/2024.

[10] *LLVM for Grad Students*. `https://www.cs.cornell.edu/~asampson/blog/llvm.html`, last accessed on 6/4/2024.

[11] *LLVM Programmer’s Manual*. `https://llvm.org/docs/ProgrammersManual.html`, last accessed on 6/4/2024.

[12] *LLVM Programmer's Manual: The Core LLVM Class Hierarchy Reference*. https://llvm.org/docs/ProgrammersManual.html#the-core-llvm-class-hierarchy-reference, last accessed on 14/4/2024.

[13] *Clang command line argument reference*. https://clang.llvm.org/docs/ClangCommandLineReference.html#actions, last accessed on 6/4/2024.

[14] *llvm-dis - LLVM disassembler*. https://llvm.org/docs/CommandGuide/llvm-dis.html, last accessed on 29/4/2024.

[15] Reema Thareja. "Data Structures Using C". In: 2nd. Oxford University Press, 2014, pp. 46, 163, 164, 283.

[16] Richard O. Kirk et al. "Warwick Data Store: A Data Structure Abstraction Library". In: *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2020, p. 72. DOI: 10.1109/PMBS51919.2020.00013.

[17] *C reference: Array declaration*. https://en.cppreference.com/w/c/language/array, last accessed on 7/4/2024.

[18] *C reference: Pointer declaration*. https://en.cppreference.com/w/c/language/pointer, last accessed on 7/4/2024.

[19] *C reference: Dynamic memory management*. https://en.cppreference.com/w/c/memory, last accessed on 7/4/2024.

[20] *C Reference: Objects and alignment*. https://en.cppreference.com/w/c/language/object#Alignment, last accessed on 14/4/2024.

[21] *LLVM Opaque Pointers*. https://llvm.org/docs/OpaquePointers.html, last accessed on 15/4/2024.

[22] *Structure Layout Optimizations in the Open64 Compiler: Design, Implementation and Measurements*. https://www.capsl.udel.edu/conferences/open64/2008/Papers/111.pdf, last accessed on 14/4/2024.

[23] *Intel Corporation: Memory Layout Transformations*. https://www.intel.com/content/www/us/en/developer/articles/technical/memory-layout-transformations.html, last accessed on 14/4/2024.

[24] *llvm::GetElementPtrInst Reference*. https://llvm.org/doxygen/classllvm_1_1GetElementPtrInst.html, last accessed on 19/4/2024.

[25] *llvm::AllocaInst Class Reference*. https://llvm.org/doxygen/classllvm_1_1AllocaInst.html, last accessed on 20/4/2024.

[26] *llvm::ArrayType Class Reference*. https://llvm.org/doxygen/classllvm_1_1ArrayType.html, last accessed on 7/4/2024.

[27] *llvm::Module Class Reference*. https://llvm.org/doxygen/classllvm_1_1Module.html, last accessed on 20/4/2024.

[28] *llvm::GlobalVariable Class Reference*. https://llvm.org/doxygen/classllvm_1_1GlobalVariable.html, last accessed on 20/4/2024.

[29] *llvm::Type Class Reference*. https://llvm.org/doxygen/classllvm_1_1Type.html, last accessed on 20/4/2024.

[30] *LLVM Programmer's Manual: The isa<>, cast<> and dyncast<> templates*. https://llvm.org/docs/ProgrammersManual.html#the-isa-cast-and-dyn-cast-templates, last accessed on 20/4/2024.

[31] *C reference: calloc*. https://en.cppreference.com/w/c/memory/calloc, last accessed on 20/4/2024.

[32] *C reference: malloc*. https://en.cppreference.com/w/c/memory/malloc, last accessed on 20/4/2024.

[33] *llvm::LoadInst Reference*. https://llvm.org/doxygen/classllvm_1_1LoadInst.html, last accessed on 20/4/2024.

[34] *llvm::StructType Class Reference*. https://llvm.org/doxygen/classllvm_1_1StructType.html, last accessed on 21/4/2024.

[35] *llvm::DataLayout Class Reference*. https://llvm.org/doxygen/classllvm_1_1DataLayout.html, last accessed on 23/4/2024.

[36] *Slurm Workload Manager - sbatch*. https://slurm.schedmd.com/sbatch.html, last accessed on 27/4/2024.

[37] *time(1) — Linux manual page*. https://man7.org/linux/man-pages/man1/time.1.html, last accessed on 27/4/2024.

[38] *High Performance Computing Job Management - The "seff" command (Slurm Job Efficiency Report)*. https://hpc.nmsu.edu/discovery/slurm/job-management/#_the_seff_commandslurm_job_efficiency_report, last accessed on 27/4/2024.

[39] Ian Sommerville. "Software Engineering". In: 10th. Pearson Education, 2016, p. 78.

[40] *Github: Profile page*. https://github.com/Mahiethan, last accessed on 12/4/2024.

[41] *Trello: project management tool*. https://trello.com/, last accessed on 12/4/2024.

[42] *STREAM benchmark*. https://www.cs.virginia.edu/stream/ref.html, last accessed on 29/4/2024.

[43] *ASA Advanced Supercomputing (NAS) Division: NAS Parallel Benchmarks*. `https://www.nas.nasa.gov/software/npb.html`, last accessed on 29/4/2024.

[44] *What's the difference between C++ and C?* `https://isocpp.org/wiki/faq/c#c-diffs`, last accessed on 29/4/2024.