Detection and Optimisation of Data Structures in Compile Time

CS351 Progress Report

Mahiethan Nitharsan

November 2023

Contents

1 Brief Introduction

The number of transistors in integrated circuits has doubled every two years, following Moore's Law since 1965, driving significant advancements in processor speeds. With innovations like multiprocessing, current processors are approaching their maximum physical potential. However, main memory performance has not kept pace, creating a substantial performance gap between processors and memory. Hardware and software optimisations are necessary to bridge this gap.

While hardware-based memory optimisations exist, such as making efficient use of the memory hierarchy, programs often lack code refactoring for improved data layout in memory. This deficiency can significantly impact the performance of memory-bound programs, such as real-time physics simulations. This can be resolved by applying essential optimisations directly to the source code before compilation, but this can be challenging for users without programming expertise.

Existing compiler infrastructures, most notably the LLVM project, often include an optimisation phase that inputs the source file and outputs an efficient target program using compiler passes [1]. A large variety of compiler passes exist for use in compilation to improve the efficiency and compute-bound performance of programs. However, no passes exist to directly optimise data structures, which can store large amount of data in memory and are possibly involved in memory-bound operations.

An alternative issue may arise from altering these data stores, leading to detrimental effects such as breaking the functionality of the program, introducing illegal memory accesses or resulting in inaccurate outputs. These problems would not be ideal for simulation programs that expect to run for continuous periods of time and require precise results to be produced.

This project aims to develop further from the existing LLVM library of compiler passes, and implement new compiler passes that automatically detect, identify and optimise data structures within programs, whilst also solving the aformentioned limitations.

2 Background Research

2.1 LLVM

A compiler consists of an analysis stage (front-end) and a synthesis stage (back-end). The main focus of the project stems from the synthesis stage, where target program is constructed from the intermediate representation (IR) and symbol tables. In between this construction, some compilers include an optional machine-independent optimisation phase. This phase performs transformations to the intermediate representation such that the back-end can produce an optimised target program, that would otherwise not be produced from the original optimised IR [2].

The LLVM compiler infrastructure project includes a machine-independent IR optimiser. As shown in Figure A.9 in the specification, the optimiser makes use of either analysis and/or transform passes, which traverse though portions or all of the IR to collect information or modify the program respectively [1]. The LLVM project comes with an extensive list of analysis and transform passes that are able to detect blocks of code, such as 'for' loops, and provide basic optimisations to increase efficiency, such as removing redundant operations [2]. This is what the project will develop on to create new compiler passes, that allow direct optimisations on data structures.

2.2 LLVM C++ API

There exists a C++ API for LLVM which is used to create transform and analysis passes. This API provides a thorough list of classes and interfaces to interact with the IR using C++ [3]. There is an option for using a C-interface to the LLVM libraries (LLVM-C), which would be preferable with better experience with the C language, however only parts of the LLVM library are implemented to this interface [4] so the full functionality of LLVM cannot be used. This is not ideal as it would make it difficult to create an effective compiler pass. As the C++ API classes and functions are very well documented, the minimal knowledge and experience in C++ should not pose a big risk during development.

A LLVM program can be broken down into four main components: Modules, Functions, BasicBlocks and Instructions [5]. Modules are container classes for Functions. Each Function is built up of BasicBlocks, which are continuous

chunks of Instructions. Figure 1 shows how each iteration can be iterated through, which can be implemented using nested for loops to provide the basic framework for the compiler passes.

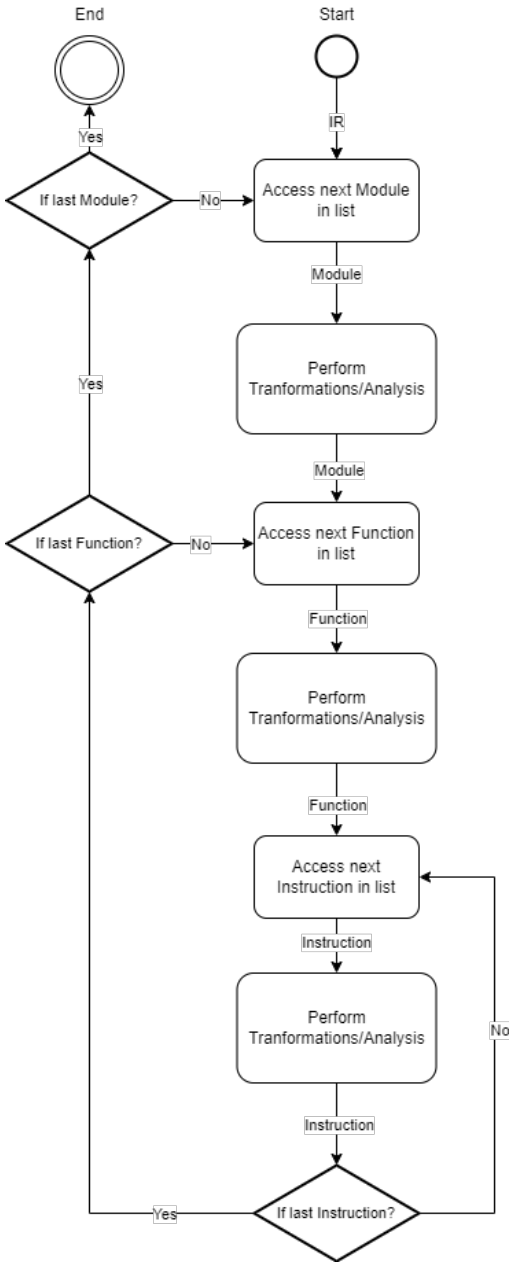

Figure 1: Iteration of all instructions

For each Module, Function, BasicBlock and Instruction instances traversed in the compiler pass, the functions belonging to those classes and inherited from superclasses would be used to implement the detection and optimisations of the data structures.

The main research on what functions and classes to used from the LLVM C++ API will be done in each iteration as the project progresses. This will be covered as technical content for each completed iteration under the section 'Planning and Research'.

2.3 Required LLVM/Clang tools

The official LLVM guide [6] and external tutorials [5] were researched to understand what LLVM tools are required to create a compiler pass and use it for optimisation.

Firstly, when a compiler pass is written in C++, it is registered with a pass name and is compiled to an .so file. This .so file is then used with the LLVM's modular optimiser and analyser tool `opt` [7] to run the compiler pass on LLVM source files. The `opt` tool consists of many flags to help with testing of the optimisation, such as measuring the time needed for each pass using the flag `-time-passes`.

The input source files can either be `.ll` files and `.bc` files, which is the LLVM IR in assembly and bitcode representations respectively. This is generated by the LLVM Clang front-end, the `clang` command with the flag `-emit-llvm` [8]. Further flags can also be used to make the LLVM IR assembly code more readable, for example, using the `-fno-discard-values` flag does not removes names of values such as pointers when generating the IR. LLVM assembly code in `.ll` files are human readable, which makes it easier to analyse and test the IR.

3 Project overview

3.1 Changes to the iterations

In the research and planning phase of Iteration 1 (which focuses on detecting Array of Structs), it was discovered that there are two different ways of declaring arrays: static and dynamic arrays.

Static arrays are the standard way of initialising arrays, which are contiguously allocated non-empty sequence of objects with a particular element type. This word 'static' refers to the fact that the array size (number of objects) do not change during its lifetime [9].

Dynamic arrays are created using pointers. Pointer declarations are used for dynamic storage, which means that they can be allocated and reallocated bytes of memory during its lifetime [10]. Pointers can be allocated a certain number of bytes such that it can store a certain number of elements of a specific type, e.g. 400 bytes can be allocated to store 100 `int` elements, each of 4 bytes.

The two weeks duration assigned to Iteration 1 is not enough to research, plan and implement the detection of both static and dynamic AoS. Thus, it was decided to split Iteration 1 into two separate iterations, to cover the detection of both of these types. The implementations from both iterations will then be combined into one compiler pass, to meet the requirement of detecting all Array of Structs in a program. The final list of iterations of this project is as follows:

Iteration 1: Detecting **static** Array of Structs

Iteration 2: Detecting **dynamic** Array of Structs

Iteration 3: Optimising Array of Structs

Iteration 4: Detecting Struct of Arrays

Iteration 5: Optimising Struct of Arrays

Iteration 6: Detecting Linked Lists

Iteration 7: Optimising Linked Lists

Iteration 8: Detecting Trees

Iteration 9: Optimising Trees

Figure 2 shows the project timetable, with the updated iterations. The agile nature of this project made it very easy to apply these changes without causing major interruptions to the whole project schedule.

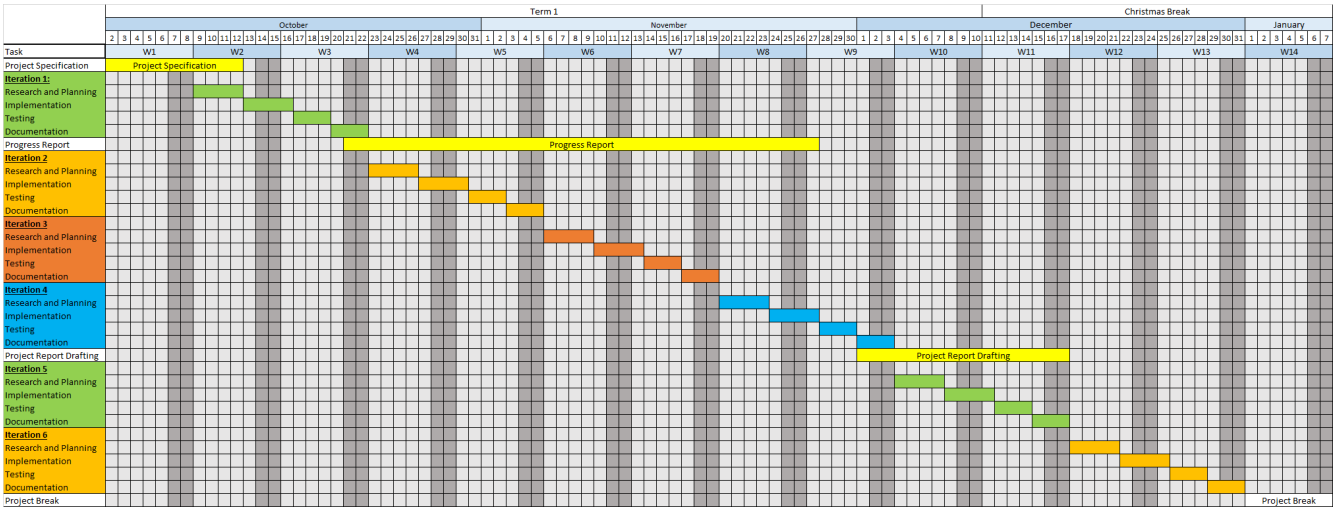

Figure 2: Updated Term 1 timetable

3.2 Current progress

In the first eight weeks of the project schedule, the first two iterations has been completed; the compiler pass for the detection of Array of Structures (static and dynamic) has been implemented.

3.3 Iteration 1: Detection of static AoS data structures

For this implementation, the pass is able all of the possible types of static AoS data structures highlighted in the following section, on one condition that the size must be greater than zero. This is done so that later optimisations are not applied to unused static AoS types, that cannot change in size later in run-time. This helps the compiler pass have a much lower runtime, by not performing optimisations on these unused AoS pointers.

3.3.1 Research and planning

When inspecting the IR, the arrays were declared using `alloca` instruction, which allocates memory on the stack frame of the calling function [11].

A Value class acts as a superclass to many other classes such as Instruction, Function and Constants [12]. A core subclass of Value is User, which defines any operands of the Value type [13], which is helpful to analyse the operand Values of certain instructions.

The `alloca` instruction that needs to be identified falls under the Instruction class [14]. Since the `alloca` instruction takes only one operand, it should fall under the UnaryInstruction class, which it does under the subclass AllocaInst.

Figure 3 shows the derived class hierarchy to the AllocaInst functions, where its defined and derived functions will be used to further analyse the `alloca` instruction.

Figure 3: Class hierarchy for AllocaInst

A static array be defined in two ways in C :

1. The allocation size is defined by a variable.

C representation:

```
struct node
{
    int a;
    int b;
    char c;
};

...

int n = 67900
struct node array[n];
```

LLVM IR representation:

```
%struct.node = type { i32, i32, i8 }

...

%2 = alloca i32, align 4
store i32 697000, ptr %2, align 4

%5 = load i32, ptr %2, align 4
%6 = zext i32 %5 to i64
%8 = alloca %struct.node, i64 %6, align 16
```

2. The allocation size is defined by an integer literal.

C representation:

```
struct node array[6790];
```

LLVM IR representation:

```
%3 = alloca [6790 x %struct.nodeOne], align 16
```

The respective Value operands `%struct.node` and `[6790 x %struct.node]`, from both `alloca` instructions in `%8` and `%3`, needs to be fetched and its type must be checked. If its type is a struct, then this array stores a number of elements of struct type, therefore it can be confirmed as a (static) AoS.

3.3.2 Implementation

To determine whether an Instruction is an instance of an AllocaInst class, the function `dyn<cast>` from the C++ API can be used. This first checks whether the instruction is an AllocaInst, and if true, it gets casted to an AllocaInst [15].

To check the type that has been allocated by the instruction, the function `Type* getAllocatedType()` can be used in the class AllocaInst [16]. This function return a Type pointer.

To distinguish between the two AoS types, an additional function `bool isArrayAllocation()` in the AllocaInst class is used to determine if there is an allocation size parameter assigned to the Allocation instruction **and** the size is greater than 1 [16]. This function helps eliminate any empty Array of Structs (with size 0) which should not be counted as an AoS. If this function returns true, the pointer is equivalent to the first AoS type, otherwise it could be the second AoS type.

If the array is the first AoS type, the actual type of the Type pointer returned by `getAllocated()` can be checked. The Type class [17] has several boolean functions to determine whether a Value is of a particular type. In this case, the function `bool isStructType()` is used to check if the allocated elements of the array are struct types.

For the second example, however, using the boolean function `isStructType()` on the Type pointer operand - `[6790 x %struct.node]` in the example above - returns `false`, since it actually belongs to the ConstantAggregate type [18]. But it is evident that the a element of struct type is being allocated.

This limitation can be resolved by using simple string analysis, by printing the type of this operand to a string variable. The variable would store the type "[6790 x %struct.node]" as a string literal, which is then used to check if it meets the following criteria:

- The string contains the characters '[' and ']'
- The string does not start with '[0 x'
  - This is used to check that the size of the AoS is not zero
- The string contains the word "struct"

- This confirms that the element type of the AoS is a struct

Given that this criteria is met for this string, it can be confirmed as a static AoS data structure.

Figure 4 shows the design of this implementation as a flowchart.

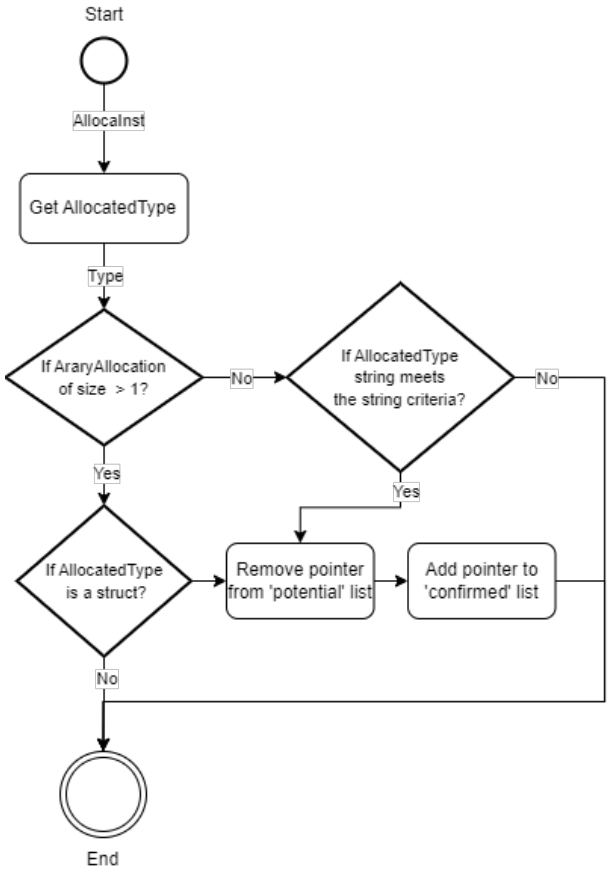

Figure 4: Static AoS detection

The two additional steps after confirming a static AoS in the figure above is a feature of the implementation in Iteration 2, since this function will be incorporated as part of that implementation.

3.4 Iteration 2: Detection of dynamic AoS data structures

For this implementation, a condition is placed such that only AoS data structures that have been accessed and/or modified the program are detected. This will not affect the optimisation process since optimising unused AoS pointers, that are not allocated memory later in the runtime, will not produce any performance improvements. Furthermore, this will help improve the performance of the optimisation passes by making sure that unnecessary optimisations are not being made.

3.4.1 Research and planning

Dynamic AoS data structures are created from pointers using dynamic memory allocation functions such as `malloc()` or `calloc()`. Inspecting the equivalent IR, this involves a `call` instruction to the respective function [19]. These instructions will be part of the CallInst class [20].

C representation:

```
struct node
{
    int a;
    int b;
    char c;
};

...

struct node* arrayOne = (struct nodeOne*) malloc(100*sizeof(struct node));
```

LLVM IR representation:

```
%struct.node = type { i32, i32, i8 }

...

%call = call noalias ptr @malloc(i64 noundef 400)
```

The size 400 (bytes) in the instruction equates to 100 `int` elements of size 4 bytes.

Similar to static AoS, the allocation size can be passed in as variable. The differences do not matter here as the detection only needs to look out for `malloc()` or `calloc()` function calls.

Since `malloc()` or `calloc()` can be used to create pointer of different types, detecting these function calls does not uniquely identify an AoS pointer. There is no way to determine whether the pointer used by the function call is a struct type, since the current LLVM project version has replaced typed pointers with opaque (untyped) pointers to solve semantic issues [21].

Global variables can also be a dynamic AoS. In C, these are allocated memory inside functions in the same way as local variables, but are declared outside of function bodies.

LLVM IR representation of global AoS variables:

```
@globalOne = dso_local global ptr null, align 8
```

3.4.2 Implementation

All malloc-ed pointers are to be stored in a 'potential' list. Each function call must be checked to see if it reads from/writes to a pointer memory that is stored in this list of potential AoS types. It is decided that only functions that have been called within the `main()` function are going to be inspected instead of checking all functions in the program, in order to make the compiler pass run more quickly, especially when analysing large programs. The names of called functions are obtained from the operands of CallInst instructions and stored in another list, so each one can be inspected independently.

When inspecting each function, the aim is to identify a `getelementptr` instruction [22] that operates on one of the pointers stored in the 'potential' list. The result element type of this GetElementPtrInst instruction is obtained using the function `Type* getResultElementType()` and checked whether is is a struct type [23]. If both of these checks pass, the matched pointer can be confirmed as a dynamic AoS, removed from the list then put into a new list called 'confirmed'. This list can then be used to obtain AoS pointers that require optimisations.

As part of extreme programming, the principle of continuous integration [24] was used to add the completed implementation of Iteration 1 into compiler pass of Iteration 2, to create one whole compiler pass that detects all types of AoS data structures.

The first implementation of the analysis pass checks function calls that are present in the `main()` function only. Other functions themselves may declare AoS pointers and make function calls to access/modify the elements, which cannot be detected by this first implementation. Therefore, additional time was spent to replan and produce the second implementation, which made sure all of the analysis were recursively when accessing each called function of the program.

Figure 5 below shows the general design of compiler pass used to detect all AoS data structures.

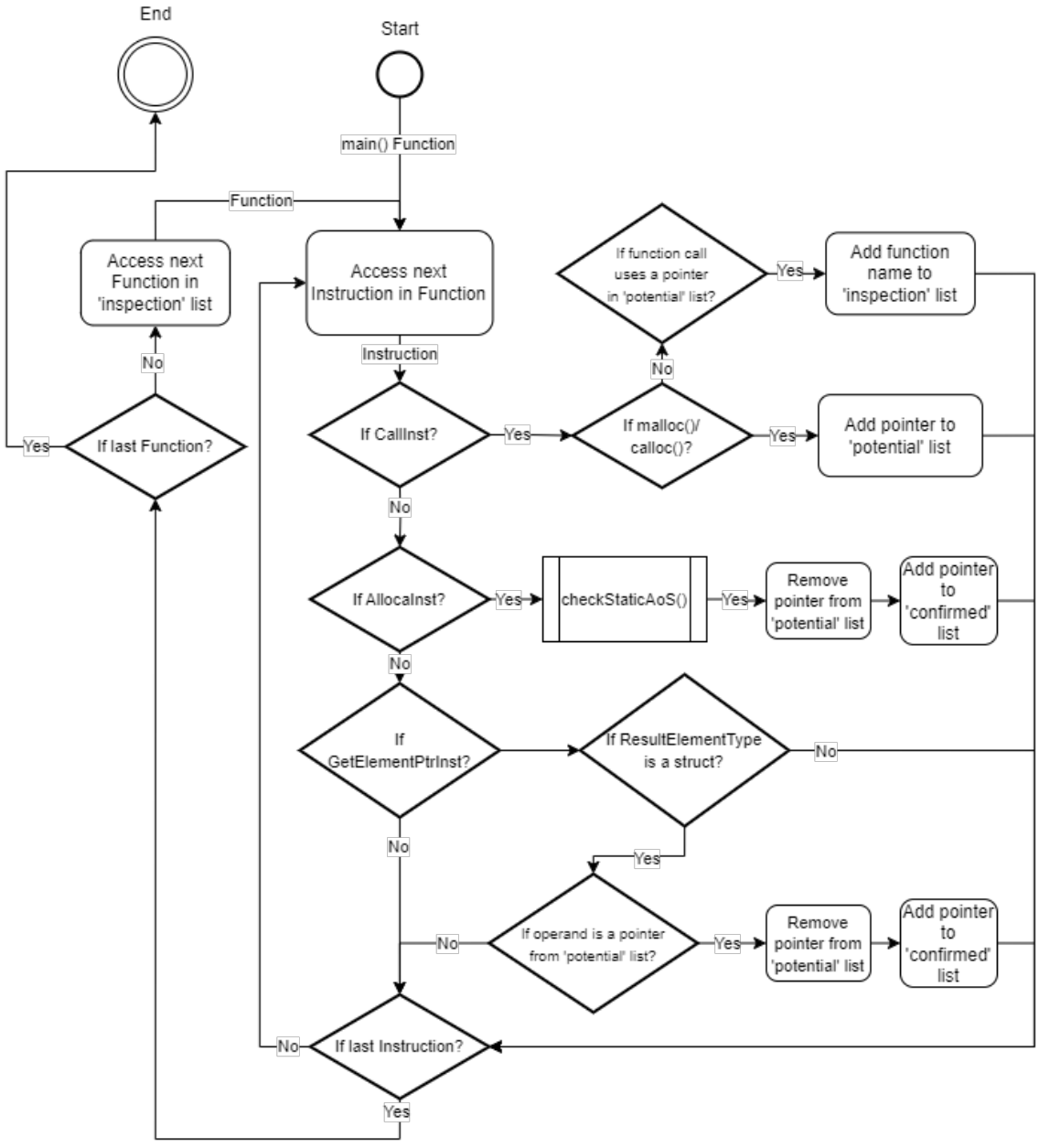

Figure 5: Detection of AoS data structures

The process `checkStaticAoS()` used in in the flowchart above is represented by Figure 4 designed in the Iteration 1 implementation phase.

3.5 Review of progress

Figure 6 shows a comparison of the actual progress against the planned timetable.

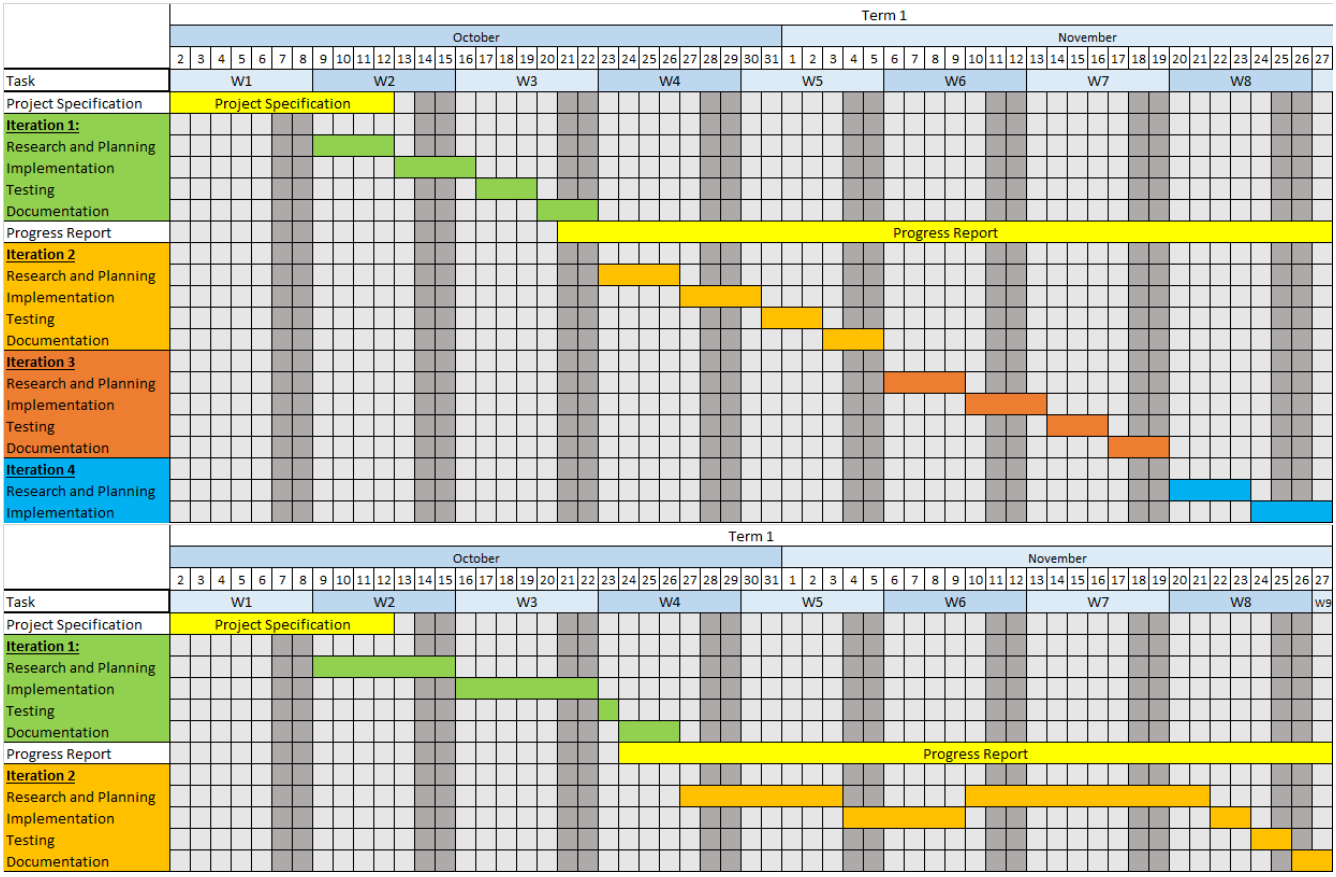

Figure 6: Comparison between planned progress (top) and actual progress (bottom)

The current project progress is delayed by three weeks when compared to the initial schedule. Iterations 1 and 2 took longer to complete than the proposed time of two weeks. Both iterations were delayed by the workload and commitments coming from external university work, such as coursework and assignments of other studied modules. Since these deliverables had significant assessment weightings, it required more study hours to complete, therefore most of time planned for this project had to be reallocated into completing these courseworks and assignments.

A main interruption in the project progress is caused by Iteration 2. The implementation phase of this iteration involved two versions of the detection pass. The main cause of this is due to poor planning. During the planning phase, all cases should have been considered for testing before designing the structure of the implementation. The first implementation was not able to detect AoS data structures declared within multiple levels of function calls, so the planning phase had to be re-instated at the Week 6 in order to fix this. Due to other coursework deadlines in Week 7, the project progress was halted such that the planning phase was only completed in Week 8.

Two solutions existed to deal with these delays: the iteration can either be skipped and finished later, or work can be continued. The former solution could have been viable, since the original project timetable had time allocated for a project break, which could have been used as a contingency plan to finish off these iterations. However the latter solution was chosen. This is because the first two iterations provide the basic framework for implementing detection passes and also provide some planning and research content, both which will aid development of following iterations. By completing these first two iterations, regardless of delays, it can allow the following iterations to be completed in a much shorter time by applying the same methods and techniques used in these two iterations.

All of this could have been averted in the first place by allocating more reasonable times to each iteration and by adding a time contingency plan in the timetable, to ensure that time is available to recover from any interruptions. In the specification, it was thought that two weeks was an overestimate for these first two iterations as there was

no consideration of any upcoming coursework and assignments from other modules. This was not the case during this term, as the coursework and assignment load increased during Weeks 5, 6 and 7, which resulted in unexpected delays in Iteration 2.

For the future plan and schedule, the times to be allocated to each iteration will be more accurate and justified based on current and upcoming work commitments in Term 2. Furthermore, some time contingencies would be added in the timetable, to ensure that in the worst case scenario, sudden delays do not interrupt the workflow and progress of the project.

In conclusion, the use of agile methodologies such as extreme programming proved beneficial in this project by producing completed software that can be integrated with the whole system very easily. Agile project management allows quick changes to these iterations, however this is only possible if the project schedule is planned carefully to consider unexpected delays. This is done by allocating reasonable times to iterations and including time contingency plans, to prevent causing interruptions when introducing these changes. Despite the delays, the amount of work done in this iteration is good enough to allow code reuse in the following iterations and should therefore make future project progression shorter and smoother.

4 Future Plan

Figure 7 details the future plan for the rest of Term 1 and Christmas Break.

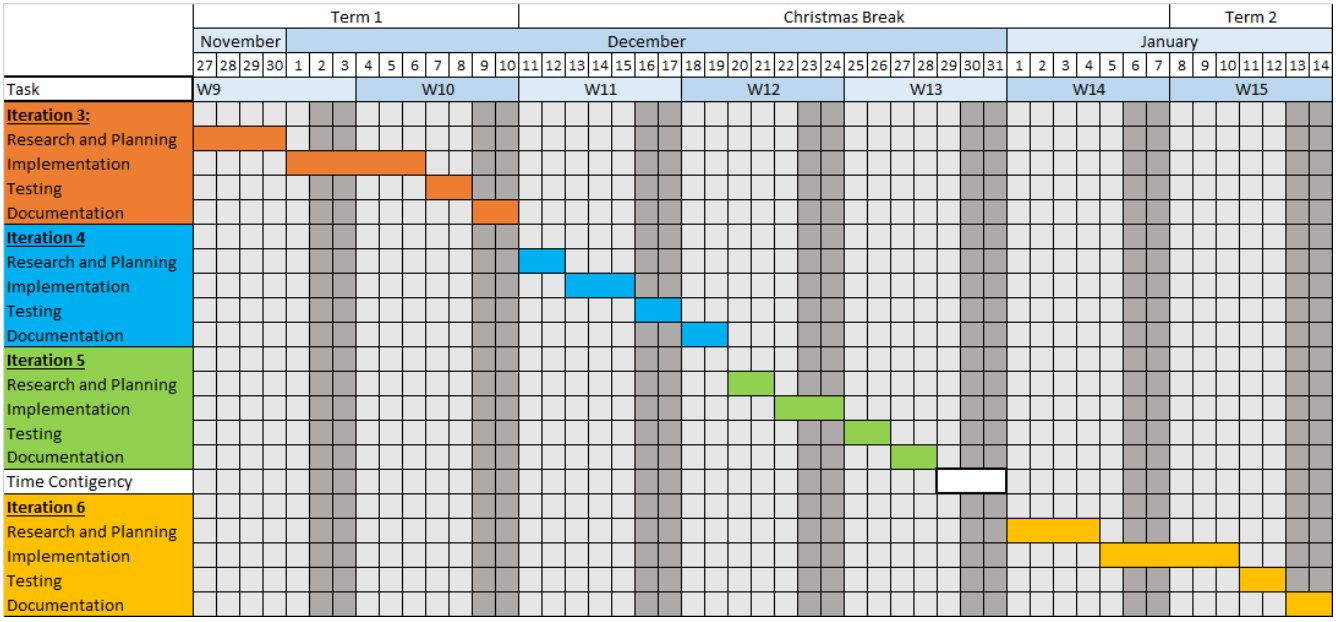

Figure 7: Future plan for Term 1 and Christmas Break

The main aim for the rest of Term 1 is to finish optimisation of AoS. This iteration will be given two weeks, which should be an overestimate given that the coursework/assignment load is now reduced.

This would lead up to the start of the Christmas Break. Iterations 4 and 5 are tasked with the detection and optimisation of Struct of Array data structures. The classes and functions involved in the detection and optimisation of AoS is very similar to SoA as well, therefore the development times for these two iterations will be short. Therefore, these iterations will be allocated nine days of development time. Considering that there would be only one coursework to complete over the Christmas Break and there is more free time available without lectures/seminars, allocated nine-days of development is justified.

If unexpected delays do occur and Iterations 4 and 5 take longer than nine days to complete, the time contingency allocated at the end of Week 13 will be used to ensure that these iterations are complete by the end of the Christmas Break. This allows only the detection and optimisations of the ADTs to be focused on throughout the whole of Term 2.

If no delays occur during the development of Iterations 4 and 5, work on Iteration 6 can begin early at Week 14, instead of taking a Project Break as proposed in the original schedule. By starting iterations as early as possible, the remaining time left at the end of project development (end of Term 2) can be used as a contingency to finish off incomplete work or recover from any unexpected delays.

Figure 8 shows the adapted timetable for Term 2.

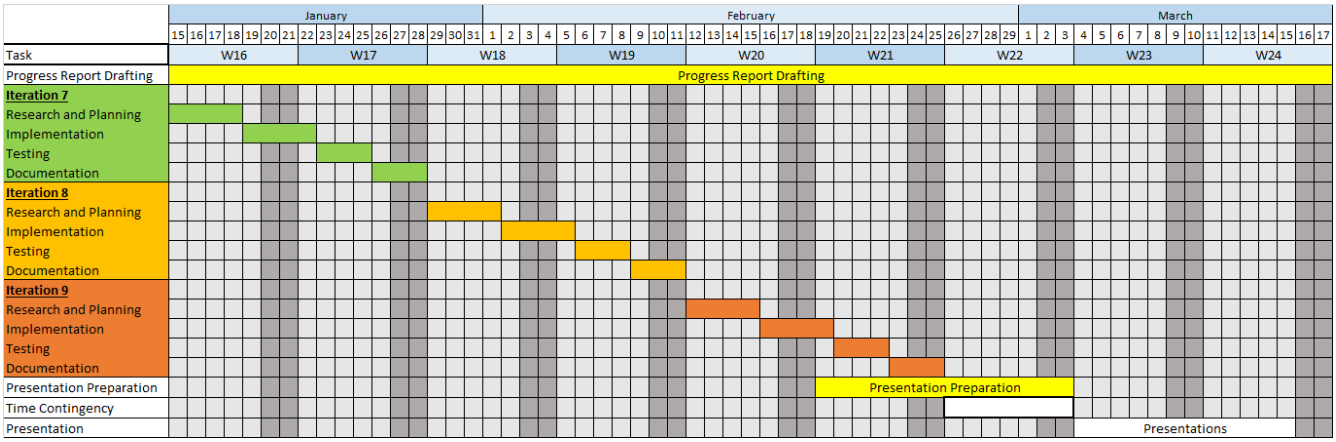

Figure 8: Future plan for Term 2

Iterations 6, 7, 8 and 9 focus on Abstract Data Types (Linked List and Trees), which may be structured quite differently to AoS and SoA types, thus may require new methods for detection and optimisation. Therefore, two-weeks of development time will be allocated to these iterations to allow for more planning and research.

Project Report Drafting is pushed to Term 2, which removes the need to parallelise this task with the iterations. This allows most of the effort to be put into the development iterations, which is essential in order to catch up with the delays in the schedule plan that occurred with Iterations 1 and 2.

For any unforeseen delays present in these last three iterations, the time contingency allocated at the end of Term 2 will be used to resolve these delays. Otherwise, this time contingency would provide other benefits such as time to improve code, focus on external commitments, prepare early on the presentation/final report or simply take a project break that was not possible during the Christmas Break.

The timetable for Term 3 in Figure A.13 of the Specification remains unchanged, given that this is post-completion and its tasks focus on writing the Project Report.

5 Ethical Considerations

As mentioned in the Specification, there are no social or ethical issues to consider in this project, since there is no work involving personal data. At this phase of the project, there are no new ethical or social issues to consider, and the project would still attempt to avoid using personal data to avoid any of these issues from arising.

LLVM still remains open-source and the project will adhere to the licensing policies under "Apache 2.0 License with LLVM exceptions". This licence ensures that the project can freely use LLVM for personal purposes. This licence enforced the requirement to retain the copyright notice during redistribution of LLVM [25]. This does not apply to the project since the compiler passes does not include parts of the LLVM infrastructure and will not be distributed after completion.

A professional issue considered in the Specification is plagiarism, which was considered throughout the planning and research phases of project development. Effort was made to prevent this issue by noting down sources and references on Trello, the productivity tool use for the management of this project [26]. Even though LLVM is open-source and does not impose the need for proper citations, this issue mainly focues on other sources, such as textbooks or online tutorials. Any uses of these sources will be properly cited in this Progress Report, and the upcoming Presentation and Final Report.

References

[1] LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html, last accessed on 24/11/2023.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.*, pages 4–5, 583. Pearson Education, pearson new international edition, 2013.

[3] LLVM Programmer's Manual. https://llvm.org/docs/ProgrammersManual.html, last accessed on 24/11/2023.

[4] LLVM-C: C interface to LLVM. https://llvm.org/doxygen/group__LLVMC.html, last accessed on 24/11/2023.

[5] LLVM for Grad Students. https://www.cs.cornell.edu/~asampson/blog/llvm.html, last accessed on 25/11/2023.

[6] Writing an LLVM pass. https://llvm.org/docs/WritingAnLLVMPass.html, last accessed on 25/11/2023.

[7] opt - LLVM optimiser. https://llvm.org/docs/CommandGuide/opt.html, last accessed on 25/11/2023.

[8] Clang command line argument reference. https://clang.llvm.org/docs/ClangCommandLineReference.html#actions, last accessed on 24/11/2023.

[9] C reference: Array declaration. https://en.cppreference.com/w/c/language/array, last accessed on 23/11/2023.

[10] C reference: Pointer declaration. https://en.cppreference.com/w/c/language/pointer, last accessed on 23/11/2023.

[11] LLVM Language Reference Manual: 'alloca' Instruction. https://llvm.org/docs/LangRef.html#alloca-instruction, last accessed on 24/11/2023.

[12] LLVM Language Reference Manual: The Value class. https://llvm.org/docs/ProgrammersManual.html#the-value-class, last accessed on 25/11/2023.

[13] LLVM Language Reference Manual: The User class. https://llvm.org/docs/ProgrammersManual.html#the-user-class, last accessed on 25/11/2023.

[14] llvm::Instruction Class Reference. https://llvm.org/doxygen/classllvm_1_1Instruction.html, last accessed on 25/11/2023.

[15] 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 26/11/2023.

[16] llvm::AllocaInst Class Reference. https://llvm.org/doxygen/classllvm_1_1AllocaInst.html, last accessed on 25/11/2023.

[17] llvm::Type Class Reference. https://llvm.org/doxygen/classllvm_1_1Type.html, last accessed on 25/11/2023.

[18] llvm::ConstantAggregate Class Reference. https://llvm.org/doxygen/classllvm_1_1ConstantAggregate.html, last accessed on 26/11/2023.

[19] LLVM Language Reference Manual: 'call' Instruction. https://llvm.org/docs/LangRef.html#call-instruction, last accessed on 26/11/2023.

[20] llvm::CallInstClass Reference. https://llvm.org/doxygen/classllvm_1_1CallInst.html, last accessed on 26/11/2023.

[21] LLVM Opaque Pointers. https://llvm.org/docs/OpaquePointers.html, last accessed on 26/11/2023.

[22] LLVM Language Reference Manual: 'getelementptr' Instruction. https://llvm.org/docs/LangRef.html#getelementptr-instruction, last accessed on 26/11/2023.

[23] llvm::GetElementPtrInst Reference. https://llvm.org/doxygen/classllvm_1_1GetElementPtrInst.html, last accessed on 26/11/2023.

[24] Ian Sommerville. *Software Engineering*, page 78. Pearson Education, 10th edition, 2016.

[25] LLVM Developer Policy: New LLVM Project License Framework. `https://llvm.org/docs/DeveloperPolicy.html#new-llvm-project-license-framework`, last accessed on 25/11/2023.

[26] Trello: project management tool. `https://trello.com/`, last accessed on 25/11/2023.

Appendix A Specification

A.1 Problem Statement

The number of transistors in integrated circuits has been doubling every two years [1], closely following Moore's Law since its inception in 1965. As a result, we have seen huge advancements in processor speeds. In addition to new technologies such as multiprocessors and data-level parallelism, current-day processors are now reaching their maximum physical potential for improvement. Meanwhile, the performance of main memory in computers has not improved to the same level as processors. This means there exists a huge performance gap [1] between processor and memory speeds such that the main memory cannot keep up with the fast CPU requests. Therefore, hardware and software optimisations must be made to bridge this performance gap between the processor and main memory.

Even though hardware-based memory optimisations exist, such as the use of the memory hierarchy [1], programs often lack code refactoring that can improve the data layout in memory. This will have a detrimental effect on the performance of memory-bound programs, for instance, physics simulations, which require fast and consistent runtime performance to produce reliable data in a continuous format. The necessary optimisations can be applied directly to the source code before compilation. However, users of these programs (physicists or engineers) may lack the programming expertise to optimise the code themselves manually. Furthermore, certain data structures that are built using many lines of code would be tedious and difficult to modify by hand without making any errors. A better solution is to automatically apply memory optimisations during compile time without user involvement. This way, the user audience can obtain fast, continuous results from these programs, without worrying about any performance issues that could prevent them from receiving the intended results.

This project aims to detect and identify data structures within programs, which serve as the primary stores of data in memory, and apply various optimisation techniques to enhance memory usage, leading to better runtime performance. Applying optimisations during compile-time is much more convenient than doing it directly to the source code. However, the challenge lies in ensuring that the optimisations do not break the program and do not alter any outputs. This is particularly crucial in simulation programs that require precise results. Therefore, the project must also aim to optimise the program without affecting its functionality, whilst ensuring that the optimisation methods work independently and also in harmony with each other without any conflicts.

A.2 Objectives

This project is broken down into two primary objectives: detection and optimisation. Each objective is broken down into smaller sub-objectives and is given priorities (Must, Should, Could, Won't) based on the MoSCoW prioritisation technique [2]:

R1. Detect data structures within a program.

- The project **MUST** identify which data structure is being used.
  - This can be measured by checking for specific flags/tags, that enclose the data structure in the source code.

- The project **MUST** be able to detect simple data structures such as Array of Structs (AoS) and Struct of Arrays (SoA).
  - The project **MUST** be able to detect Array of Structs (AoS) by the end of Week 3.
  - The project **MUST** be able to detect Struct of Arrays (SoA) by the end of Week 7.

- The project **MUST** be able to detect abstract data structures such as linked lists and trees.
  - The project **MUST** be able to detect Linked List data structures by the end of Week 11.
  - The project **MUST** be able to detect Tree data structures by the end of Week 18.
  - Most programs are complex and make use of abstract data types (ADT) like linked lists and trees. Therefore, it is essential to identify these data structure types for optimisation. The project will focus on detecting Linked List and Tree data structures. However, as an extension, the project **COULD** cover other ADTs such as hash maps. The requirements will be adapted if this is the case.

- The project **SHOULD** be able to detect data structures used in the program within 30 seconds.
  - The detection of each data structure should be done within a reasonable amount of time, depending on the size of the program.

- The project **WON'T** identify functions that use data structures, such as traversals, searching or insertion and deletion functions.
  - Though this can be used to apply optimisations to compute-bound code, this does not directly optimise data structures and their memory usage.

R2. Optimise the data structure that is detected within the program.

- The project **MUST** apply appropriate optimisations based on the detected data structure.
  - Optimisations **MUST** be applied to Array of Structs (AoS) by the end of Week 5.
  - Optimisations **MUST** be applied to Struct of Arrays (SoA) by the end of Week 9.
  - Optimisations **MUST** be applied to Linked List data structures by the end of Week 16.
  - Optimisations **MUST** be applied to Tree data structures by the end of Week 20.

- The project **MUST** implement a variety of optimisations techniques for each data structure type.
  - The optimisation methods will be discovered in the research and planning stage of each iteration.

- Each optimisation made on a data structure **MUST** ensure functional correctness.
  - The optimised program **MUST** generate the same output as the base, unoptimised program by comparing both outputs and checking for equality.

- Observations **MUST** be made on the improvement of runtime performance.
  - Each optimisation **SHOULD** aim to improve the performance by at least 5%.
  - This will be measured by comparing the runtime of the optimised program against the unoptimised program and calculating the time difference between the two runtimes. The level of improvement depends on the size of the data structure being optimised.
  - As an extension, the optimisations **COULD** be tested against a large simulation software. The specific software to test will be identified later in the project once all unit tests have been passed.

- Observations **COULD** be made on the improvement in memory usage.
  - If reliable methods can be found to analyse the memory usage of a program, this is a possible extension to the current objective.

- The project **WON'T** apply optimisations to functions that use these data structures.

A.3 Methodology

A.3.1 Project management

An agile methodology will be primarily used since this allows development to begin straightway, without initially making a whole plan of the project like in plan-driven methodologies. Agile also allows incremental development, where each iteration of the software is planned and tested individually before moving on to the next iteration.

However, it is favourable to include aspects of plan-driven methodologies in this project as well because this project requires a lot of documentation to be produced, such as the progress report and the final report. Therefore, features

of agile will allow for incremental development of the software, with each increment having independent planning and testing phases, whilst features from plan-driven methodologies will be used to document what happened at each development iteration in the progress report and final report. This means that a documentation phase will be added to each iteration, something that is not standard in agile approaches.

The specific agile technique that will be used is extreme programming (XP), an approach based on incremental development and includes principles such as test-first development and simple design [3]. The idea of incremental development will be used for this project, which will help break down the project into small stages/iterations throughout the six-month development period. Each iteration will take a maximum time of two weeks to complete and will involve:

1. Research and Planning
   - This involves gathering the necessary information and knowledge from available resources and planning out the design steps of the implementation, making sure it follows the simple design principle such that it only meets the current requirement and nothing more.
   - This involves identifying optimisation techniques to apply to the data structures.
   - Unit tests are planned and written before the implementation, as part of test-first development.
     - The unit tests for the detection phase should ensure that each data structure is detected correctly.
     - The unit tests for optimisations should ensure that there is an improvement in performance after applying the optimisation.

2. Implementation
   - The code is implemented based on the design plan.
   - Code refactoring should be done to make sure that the code is simple and maintainable.

3. Testing
   - The code is integrated into the system and all unit tests are used to test the whole project code. If successful, the next stage of the iteration can begin. Otherwise, re-visit the implementation stage to fix the errors and test again. This will repeat until the test is successful.

4. Documenting
   - At this stage, the steps taken to plan, implement and test in this iteration should be written down briefly in separate documents. This shows what progress is made and what technical methods were used, which will help when writing the progress report and final report.

The number of iterations and the requirement of each iteration will be detailed in the project timetable.

To manage tasks and keep track of current progress against the timetable, an online kanban board website called Trello [4] will be used to create "To Do" and "Completed" lists, manage notes for each iteration and separate each iteration tasks into stages to make it easier to manage and complete. Weekly meetings with the project supervisor will help give feedback on current progress and advice on what to do next.

A.3.2 System design

The project will be based around LLVM, an infrastructure of compiler tools written using C++. Figure A.9 highlights the LLVM infrastructure, which is built up of three main stages: the front-end, middle-end and back-end. The front end converts source code to a portable, language-independent intermediate representation (LLVM IR). Clang [5] converts C code to LLVM IR (either in human-readable assembly language format or binary bitcode format).

The middle-end and back-end of LLVM operate on this IR of the input program, each stage performing code optimisations/transformations and machine code generation respectively. The middle-end of LLVM goes through several compiler passes to undergo analysis or transformations. For optimisation, the IR will go through transformation passes that alter the IR code to make it perform faster and more efficiently. The final back-end stage of LLVM converts this optimised IR to native assembly code, which can be converted to a native executable file. This executable is now an optimised version of the input program that can be run with better performance.

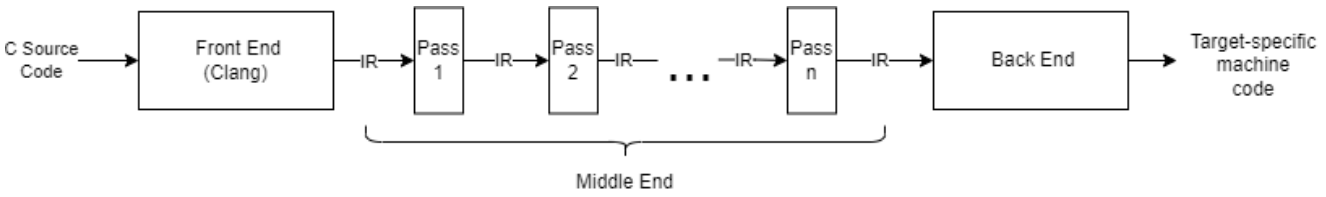

Figure A.9: Simplified LLVM compiler architecture

This project's focus is on the middle-end of the LLVM infrastructure. Multiple compiler passes will be created in this project so that the program code will pass through to detect and optimise the data structures within. These compiler passes will be developed in C++17. The LLVM front-end can work with many languages but for this project, the C programming language (C18) will be used to write the source code of the test programs containing data structures.

To achieve the main goals of this project, four compiler passes will be implemented, each focusing on detecting and optimising a particular data structure in the input program. The aim of each compiler pass is as follows, executed in order from top to bottom:

Pass 1 - Detect and optimise Array of Structs (AoS).

Pass 2 - Detect and optimise Struct of Arrays (SoA).

Pass 3 - Detect and optimise Linked List data structures.

Pass 4 - Detect and optimise Tree data structures.

Figure A.10 shows the overall design view of the project, starting from the input of the C program, applying optimisations to the IR, and receiving the optimised program as output. Additional compiler passes may be included to detect and optimise other abstract data structures, previously mentioned as an objective extension (add ref to part of doc). These would be easily added to the design, after the fourth compiler pass. The sequential design for the compiler passes also allows for modifications to the order of passes, as a solution to solve any errors or incompatibility issues.

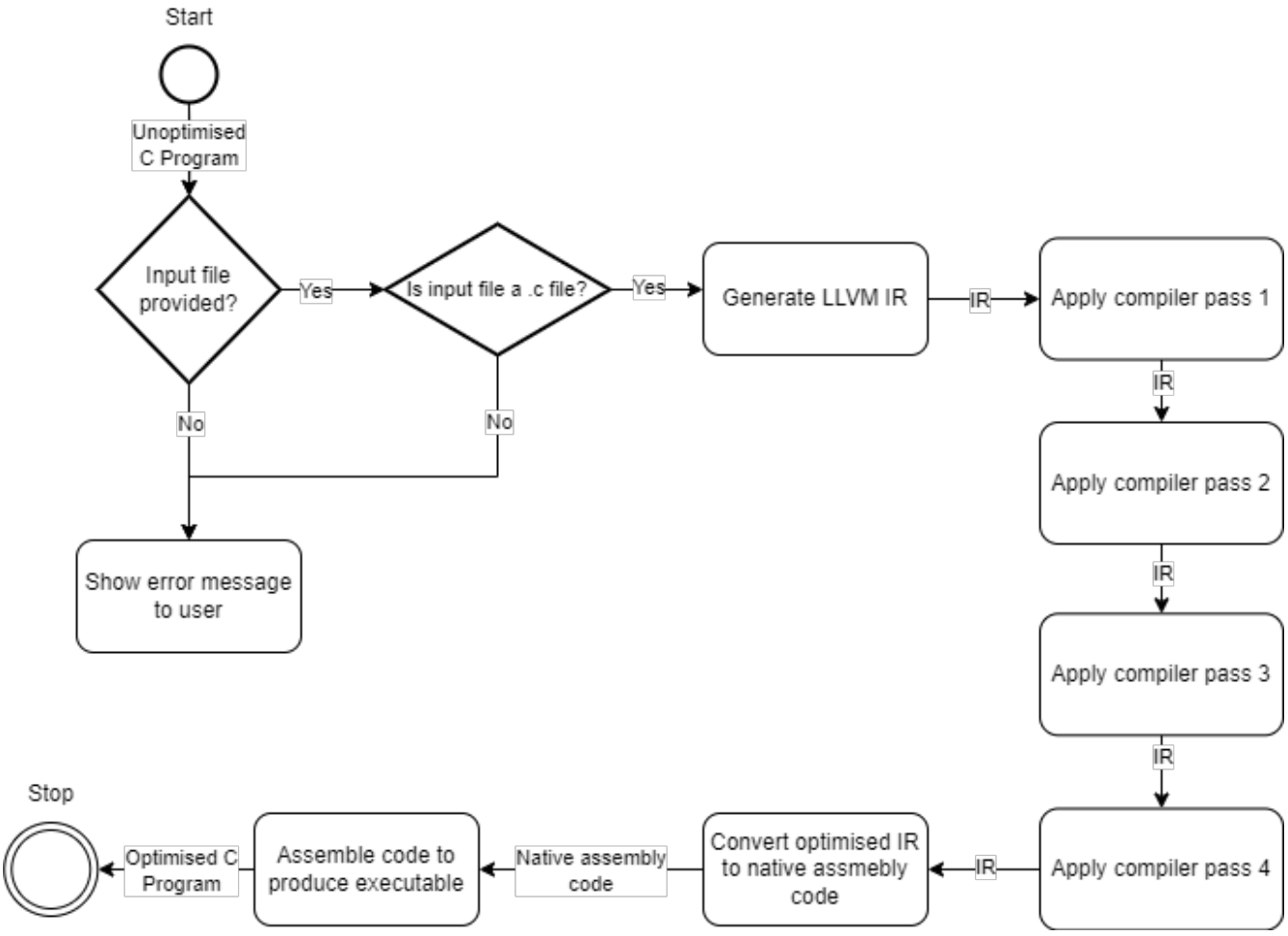

Figure A.10: System design view

A.4 Resources

A.4.1 Hardware

- Code implementation and testing will be done on a personal laptop.
  - Processor: Intel Core i7-1165G7 @ 2.80GHz
  - RAM: 16GB LPDDR4x @ 4257MHz
  - Storage: 256 GB PCIe M.2

A.4.2 Software

- LLVM (version 18.0.0git) [6]
  - Includes Clang front-end for the LLVM project (version 18.0.0) [7]
- Running LLVM and compiling programs on Windows Subsystem for Linux, running distribution of Ubuntu 22.04.3 LTS [8]
- GitHub and git version control [9]
  - This will be used to keep a backup of the project on the cloud, as well as manage versions of the project after completing each iteration during implementation.
- Visual Studio Code – version 1.83.0 (September 2023) [10]

– Used as the source code editor for C and C++ files.

A.4.3 Programming languages

- C++17 - writing the compiler passes.
- C18 - developing the unit tests; sample programs that contain data structures.

A.4.4 Documentation and tutorials

- LLVM Documentation (for version 18.0.0git) [11]

Since this is a new field of study and with no prior experience in writing compiler passes, it would be necessary to read through documentation, reports and tutorials related to using LLVM and writing compiler passes. More of this documentation and tutorials will be discovered during the research and planning phase of each development iteration, so the "Resources" section will be appropriately updated later as the project progresses.

A.5 Risks

Risk	Impact	Likelihood	Mitigation
Unable to discover the optimisation techniques in the research phase to apply to data structures.	**High**	**Low/Medium** Documentation may be available but could be difficult to understand and gather information from.	Reduce the scope of the objectives. Change objectives to focus on optimising other parts of the code. Aim to finish the other objectives first: the detection of data structures.
Difficulty in identifying and fixing C++ errors when writing the compiler passes.	**Medium**	**Medium** First time developing a project using C++ so unknown errors are likely to occur.	Read C++ language documentation extensively to find the source of error. Use developer forums such as Stack Overflow [12] for code-related assistance.
Hardware failure – laptop failing to function, no way to recover data on the built-in storage device.	**Low/Medium** The project data can be quickly recovered from the GitHub repo but reinstalling local software such as LLVM, VS Code and Ubuntu is time-consuming and will halt progress.	**Low**	Use the desktop computer at home address and move it to term-address at an appropriate time. Meanwhile, work on DCS machines at campus.
Illness during project development causing unexpected delays and progress stalling.	**Low**	**Low**	Events on the timetable are overestimated, to accommodate for any unexpected delays and changes.
New up-to-date versions of software available – applies to LLVM, VSCode and WSL.	**Low** These updates do not take too long to apply and will not affect existing project.	**Medium/High** Likely to have multiple new versions of the software during the project's six-month development.	Automatic updates are disabled, and the software versions are kept the same. Updates will only be made if a new useful feature is introduced.

A.6 Legal, Social, Ethical and Professional Issues and Considerations

The LLVM software used in this project is open-source and licenced under "Apache 2.0 License with LLVM exceptions", which permits free download and use of the LLVM framework software for personal purposes [13].

There are no social or ethical issues to consider in this project, as it does not collect and use any personal data from individuals. A professional issue that must be considered throughout the whole project is plagiarism, which must be avoided to show the honesty and integrity of the work put into this project. This can be mitigated by ensuring that references to online and book resources are correctly cited in the documentation. The ACM Code of Ethics and Professional Conduct [14] lists a set of principles that will be closely adhered to throughout the project to prevent any other professional and ethical issues from arising.

A.7 Project Timetable

The Gantt charts, shown in Figures A.11, A.12 and A.13, show a rough outline of how the project is broken down into iterations, with the project deliverables interleaved. The project will be broken down into eight iterations. Each iteration aims to implement the main objectives of the project; the project must detect and optimise four different types of data structures. The topics of the iterations are as follows:

Iteration 1: Detecting Array of Structs

Iteration 2: Optimising Array of Structs

Iteration 3: Detecting Struct of Arrays

Iteration 4: Optimising Struct of Arrays

Iteration 5: Detecting Linked Lists

Iteration 6: Optimising Linked Lists

Iteration 7: Detecting Trees

Iteration 8: Optimising Trees

The project deliverables are written and drafted in parallel with the iteration cycles, especially during the specification stages of each iteration. The methods, progress and results collated in each iteration cycle are used to help write the progress report, presentation, and final report.

In the timetables, each iteration is given a maximum of two weeks for completion. For each iteration, four days are spent on research and planning, four days on implementation, three days on testing and three days on documentation. The time allocated to each iteration is an overestimate since some iterations may take less than two weeks to complete. In such instances, the remaining time can be allocated to work outside of the project, such as coursework from other course modules.

This project timetable will be adapted over time, to show the actual duration of completion for each iteration and to show any changes in the start and end times of the timetable events.

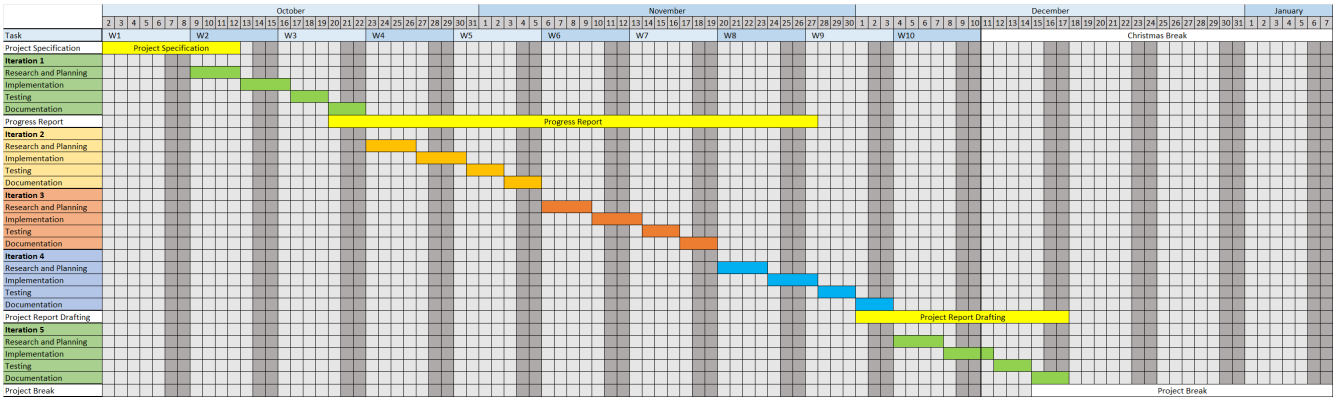

Figure A.11: Term 1 and Christmas Break timetable

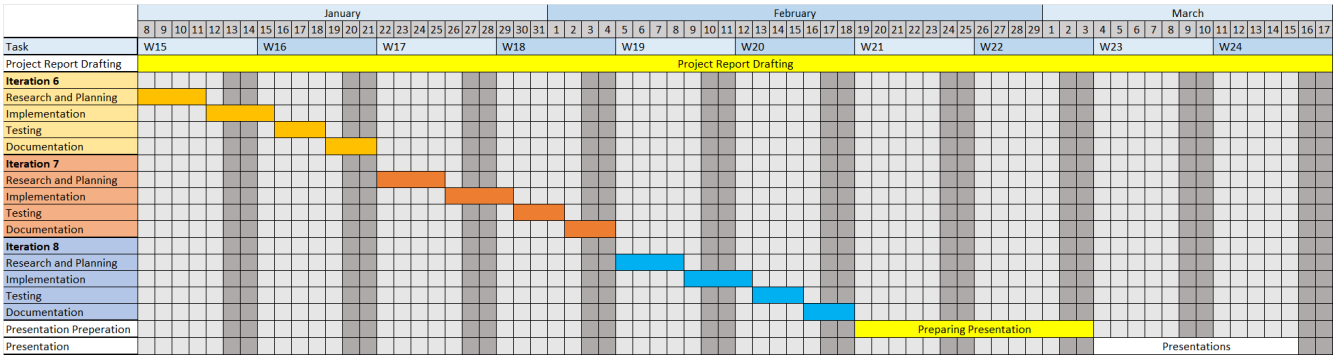

Figure A.12: Term 2 timetable

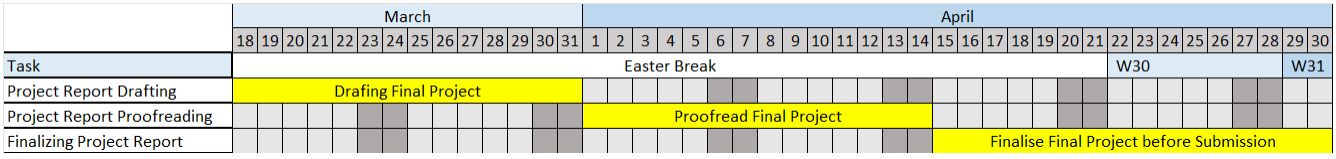

Figure A.13: Easter break and Term 3 timetable

References

[1] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*, pages 19,78,80. Morgan Kaufmann Publishers Inc., 6th edition, 2017.

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

[3] Ian Sommerville. *Software Engineering*, page 78. Pearson Education, 10th edition, 2016.

[4] Trello. `https://trello.com/`.

[5] Clang: a C language family frontend for LLVM. `https://clang.llvm.org/`.

[6] LLVM 18.0.0git Release Notes. `https://llvm.org/docs/ReleaseNotes.html`, last accessed on 10/10/2023.

[7] Clang 18.0.0git documentation, Clang Compiler User's Manual. `https://clang.llvm.org/docs/UsersManual.html`, last accessed on 10/10/2023.

[8] Ubuntu on WSL. `https://ubuntu.com/wsl`, last accessed on 10/10/2023.

[9] Github. `https://github.com/`, last accessed on 10/10/2023.

[10] Visual Studio Code updates - September 2023 (version 1.83). `https://code.visualstudio.com/updates/v1_83`, last accessed on 10/10/2023.

[11] LLVM 18.0.0git Documentation. `https://llvm.org/docs/`, last accessed on 10/10/2023.

[12] Stack Overflow. `https://stackoverflow.com`, last accessed on 12/10/2023.

[13] LLVM Developer Policy: New LLVM Project License Framework. `https://llvm.org/docs/DeveloperPolicy.html#new-llvm-project-license-framework`, last accessed on 09/10/2023.

[14] ACM Code of Ethics and Professional Conduct. `https://www.acm.org/code-of-ethics`, last accessed on 11/10/2023.