Detection and Optimisation of Data Structures in Compile Time

CS351 Specification

Mahiethan Nitharsan

October 2023

Contents

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.

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.

3 Methodology

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.

3.2 System design

The project will be based around LLVM, an infrastructure of compiler tools written using C++. Figure 1 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.

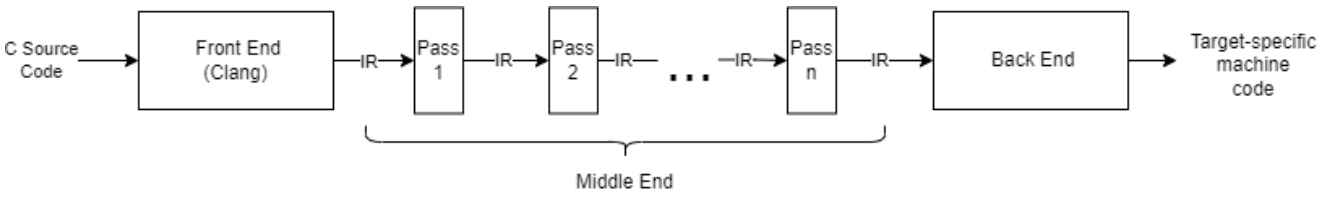

Figure 1: Simplified LLVM compiler architecture

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 2 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.

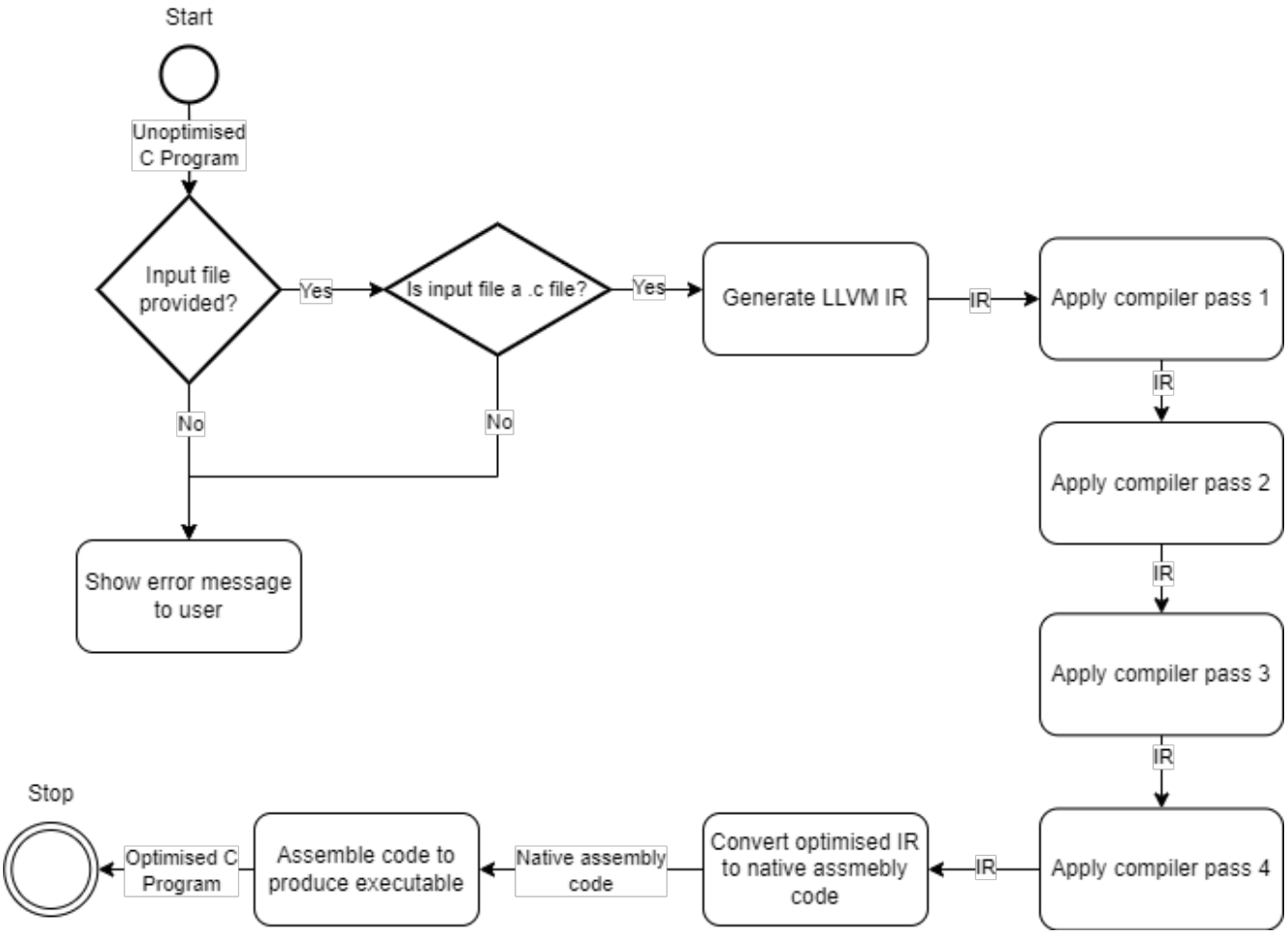

Figure 2: System design view

4 Resources

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

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.

4.3 Programming languages

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

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.

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.

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.

7 Project Timetable

The Gantt charts, shown in Figures 3, 4 and 5, 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.

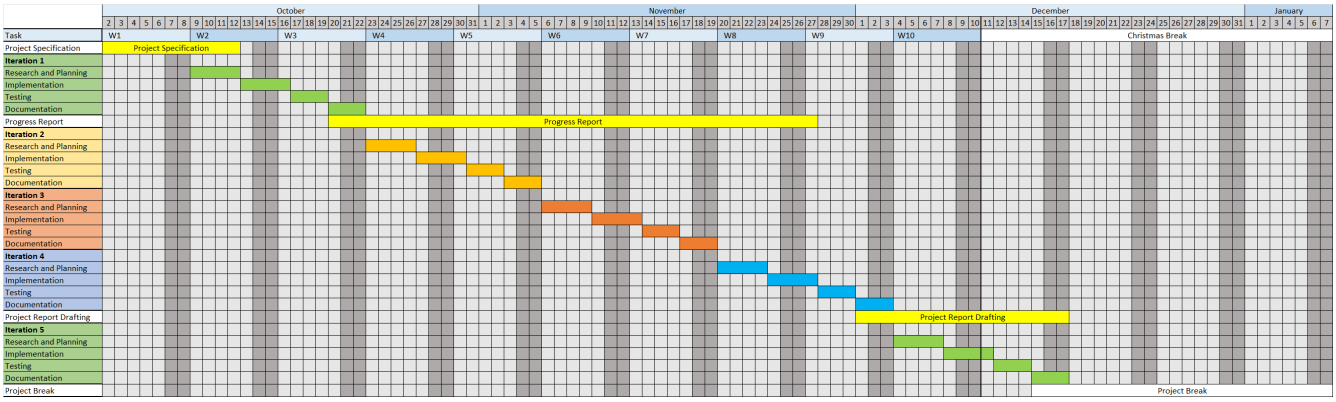

Figure 3: Term 1 and Christmas Break timetable

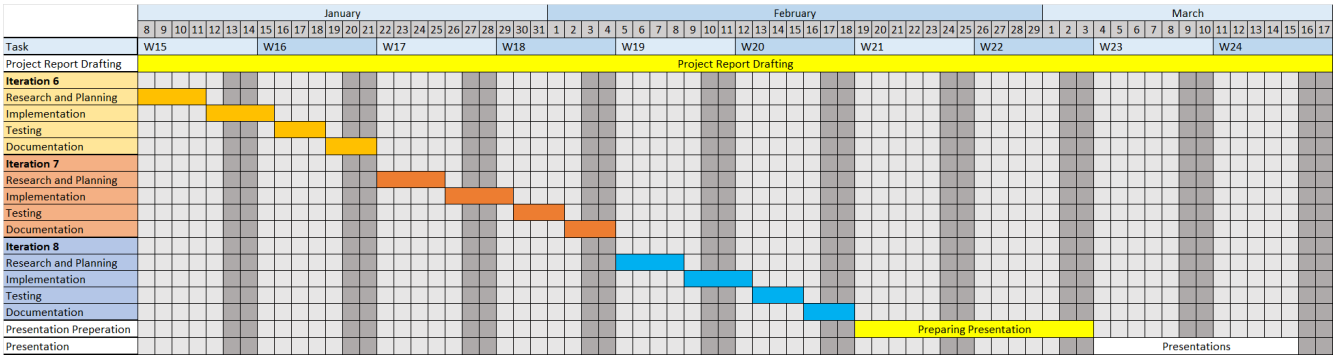

Figure 4: Term 2 timetable

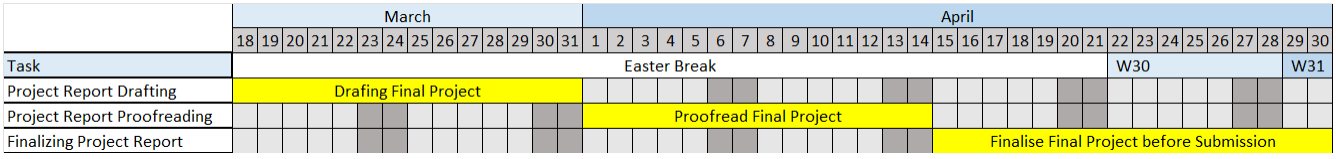

Figure 5: Easter break and Term 3 timetable

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.