6CCS3PRJ – Compiler Optimisation using AI.

Final year project

Author: Abdul Mallah

Supervisor: Dr. Andrew Coles

Student ID: 19033553

Programme: Computer Science (Artificial Intelligence)

April 12, 2024

Abstract

This project explores the use of optimisation techniques to improve performance of compiled code by manipulating GCC/G++ compiler flags, and it's application in the field of software engineering and computational optimisation; aiming to identify combinations of compiler flags that achieve reduced execution times, thus enhancing compiler efficiency and program optimisation.

In this project, we seek the use of Artificial Intelligence (Ai) to implement an optimisation framework, permitting a detailed examination of the compiler flags' configuration space, curating a selection that aims to outperform the most aggressive forms of default optimisation available. Analytical tools developed specifically for this research facilitate an objective evaluation of various flag configurations. Through a systematic combination of compiling and executing benchmark programs, alongside software-based analyses, the project derives insightful conclusions about the influence of compiler flags on execution times.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary.

I grant the right to Kings College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Signed: Abdul Rehman Mallah

Date: April 12, 2024

Acknowledgements

I would like to thank my supervisor Dr. Andrew Coles for his valuable expertise and guidance throughout this project

Contents

Introduction

In the evolving landscape of computer science and software development, compiler optimisation emerges as a pivotal area for enhancing program performance. Compilers utilise a myriad of optimisation techniques, deploying sets of compiler flags to activate these optimisations.

The GCC/G++ and clang-based compilers offer standard optimisation levels denoted by flags -O0, -O1, -O2, and -O3. Each optimisation level is a superset of the one before it [1]

It is observed that the standard optimisation level -O3 to be the overall best option as far as performance is considered with respect to execution time and instruction count [2]. The motive of this project stems from a fundamental insight: the preset combinations of optimisation flags, while may beneficial in general cases, do not guarantee optimal performance for specific code segments. In scenarios where execution speed is important, particularly in performance-sensitive programs that demand rapid response to external changes, even minor speed enhancements can lead to significant runtime reductions and, on a larger scale, a performance boost of as little as 2% can have a substantial effect on CO2 emissions. Research supports that decreased energy consumption correlates with faster code execution, highlighting the positive role of GCC/G++ optimisation in advancing green computing initiatives [3]

Moreover, performance holds paramount importance not only in critical software but also in non-critical applications that remain active, influencing broad aspects such as compilers, operating systems, and kernels. Despite this, developers often prioritise code structure, readability, and documentation, aiming for comprehensibility and maintainability. While these aspects are vital, they present opportunities to enhance optimisation further, suggesting a need to balance immediate code qualities with optimisation potential. [4]

These flags, dictating various optimisation strategies during the compilation process, can dramatically affect the performance of the resulting executable. However, not all flags contribute positively in all contexts; some may inadvertently extend execution time. By creating an algorithm to tailor compiler flags for input programs, the execution times can be significantly reduced. This reduction is achieved by minimising the overhead of flags that have little to no effect on the program's execution time, eliminating flags that may disrupt cache utilisation or parallelism, among other strategies.

Recognising this, the project aims to design and implement an algorithmic solution that intelligently navigates the vast landscape of possible compiler flag configurations. This approach addresses the challenge of manually selecting optimisation flags, an often impractical task due to the sheer number of possible combinations and the nuanced impact of each flag on different codebases.

A further objective is to validate the proposed algorithmic solution through rigorous experimentation, employing a diverse set of benchmark programs.

Background

GCC/C++ Default flags

Modern compilers are equipped with a rich array of optimisation flags, designed to enhance the execution efficiency of compiled programs. These flags, accessible to developers and compiler, provide a mechanism for fine-tuning program performance. Standard optimisation levels, such as -O0, -O1, -O2, and -O3 offer a simplified approach to this challenge by grouping flags into preset combinations that generally improve performance. However, these presets cannot guarantee optimal performance for all code scenarios. A more tailored approach, involving a careful selection and combination of individual flags, holds the potential for further optimisation. This level of customisation requires a deep understanding of the flags' functionalities and their interactions with different types of code

Examining the standard presets of compiler optimisation levels reveals distinct strategies and implications for program performance:

1. -O0
   Most optimisations are completely disabled at -O0, this stage has the advantage of allowing for fast error elimination within the application
2. -O1
   -O1 or -O would be considered the first layer of optimisations available, the compiler at this flag aims to reduce code size and execution time, without applying flags that could increase the compilation time. As it applies only simple optimisations, there is minimal risk of execution errors associated with this optimisation level. 47 flags are applied at this level
3. -O2
   -O2 is a superset of -O1. This level encompasses a broader suite of optimisations, the GCC/G++ and clang-based compilers apply nearly all supported optimisations that do not involve a space-speed trade-off. It introduces more sophisticated strategies such as instruction scheduling to expedite execution. Compilation time takes longer and uses more memory. 98 flags are applied at this level

4. -O3

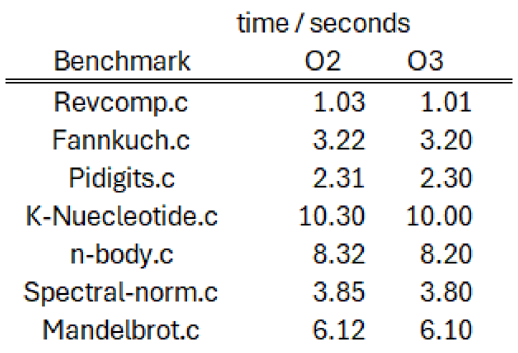
   -O3 is the most aggressive optimisation level that is the superset of -O2. As the most aggressive optimisation setting it introduces additional and more complex strategies. An example is the -floop-interchange, which reorders loop nesting to improve cache utilisation and reduce cache misses. While this level aims to deliver the fastest execution times, it may also result in larger executable sizes, a trade-off that can affect the utility of -O3 in certain contexts. [5][1]

Performance of GCC/G++ default flags

Despite -O3, plain -O2 is more common amongst developers, mainly for portability reasons and due to the aggressive nature of -O3 optimisations which can introduce risks, especially in software requiring precise floating-point operations or specific hardware interactions. Furthermore additional optimisations may require hardware support that is simply not available. Despite -O3 containing more flags there is no guarantee that it actually improves code performance, in general, developers may avoid -O3 due to larger executable being a side-effect of a performance boost that may not even be significant against -O2. [6]

Despite research and assumptions that show minimal differences between -O2 and -O3, and the prevailing preference for -O2 among developers, the pursuit of even minor speed improvements, when executed safely, can significantly benefit certain applications. Achieving these improvements while ensuring program reliability represents a critical balance in optimisation strategies.

With a focus on achieving better performance than the -O2 and -O3 flags, we will mainly be focusing on these two presets



Benchmark	time / seconds O2	O3
Revcomp.c	1.03	1.01
Fannkuch.c	3.22	3.20
Pidigits.c	2.31	2.30
K-Nuecleotide.c	10.30	10.00
n-body.c	8.32	8.20
Spectral-norm.c	3.85	3.80
Mandelbrot.c	6.12	6.10

Figure 1.0: Performance comparison between O2 and O3 against various benchmarks

Upon comparing execution times between compiling with -O2 and -O3 with a set of benchmark programs, it becomes evident that the difference in performance between -O2 and -O3 is marginal in our cases. The data indicates an average time reduction of 1.3%, this observation aligns with the expectation that while -O3 introduces more aggressive optimisation strategies, the implications on execution times for a diverse ranges of benchmark programs are often minimal.

The step up from -O1 to -O2 adds significantly more optimisation flags, 51 in total, compared to the leap from -O2 to -O3, which adds just 13. Investigating these flags introduced by -O3 shows that in general, they eliminate redundant computations, specialise functions for constant arguments, reorder nested loops for optimal cache usage, and split or merge loops to streamline execution paths. These enhancements are particularly effective for programs with tight loops doing simple math, such as data compression code.

Nonetheless, despite minimal differences in execution time between the -O2 and -O3, the latter was quicker, therefore we will be focusing on that as a starting point for our Ai algorithms.

Strategies Overview

To approach our problem we will collect all compiler flags directly from the GCC documentation, which will serve as a pool for identifying a customised set that would offer a faster execution speed than -O2/-O3.

With 182 individual flags, this problems is very complex in manifold ways, with a search space of $2^{174} = 2.4 \times 10^{52}$, we need to create an optimisation framework that will quickly converge to a solution that at least provides faster execution times than -O3 despite not being a perfect solution on all runs.

To resolve this problem we will look at various Artificial Intelligence and Machine learning techniques

Genetic algorithms

The concept of a genetic algorithm is inspired by nature; weak species are faced with extinction by natural selection, whereas the adaptability of other species allow them to survive thereby transferring their genes to subsequent generations. This process is simulated in genetic algorithms through a cycle of selection, crossover, and mutation, applied to a population of potential solutions to a given problem.

Each set of compiler flags is represented as an individual within a population. The individual's "genes" correspond to specific compiler flags, with binary gene values indicating whether a flag is activated (1) or deactivated (0). The bit-encoding of flags allow for easier tuning and direct mapping between the genetic algorithms evolutionary operations and the optimisation. [7] [8]

We must define a fitness function to measure the execution time of a individuals. In this case, a shorter execution time indicates a more desirable or “fitter” set of flags. The outcome of this function determine the likelihood for an individual’s survival and “reproduction” in the genetic algorithm process.

Once we define the fitness function and encode our flags. We can being the process:

- Start with an initial Population: Begin with a randomly generated set of compiler optimisation flags. This initial diversity is crucial for exploring a broad range of potential solutions.

- Gene representation: Each compiler flag is likened to a gene within an individual with binary values representing the flag's states.
- Crossover and mutation: New child individuals (set of flags) are created by merging portions of two or more individuals from selected parents, this selection for reproduction is based on their fitness as defined by the fitness function. Stronger or "fitter" individuals are more likely to be chosen for breeding. This crossover represents the reproduction process, creating offspring that inherit traits from multiple parents. Over successive generations ineffective flags are phased out, enhancing the population quality. The introduction of mutation rate- random alterations in the flags – introduces variability, ensuring that the algorithm explores a wide solution space thereby preventing us from sticking to local optima's.
- Convergence: The algorithm will progress until convergence, the degree of convergence could be limited based on the *n* number of iterations, *s* seconds, or performance increase. The individual with the best fitness score is selected at that point.

  Since the population size is a fixed number, the hyperparameters are mutation probability and crossover probability. Experimental analysis would be required to determine the ideal hyperparameters for our problem. [9]

  Mutation probability:

- Too High: A high mutation rate introduces excessive randomness into the population, in our case this would mean altering flags excessively. While this can prevent early convergence to local optima by ensuring constant diversity, it will also disrupt the formation of beneficial traits, leading to slower convergence and also missed out opportunities for fitter offspring, which would result in preventing the algorithm from refining a set of flags that closely approach a perfect solution
- Too Low: On the other hand, a mutation rate too low would not introduce variability and increase the likelihood of getting the solution of a local optima rather than one closer to the global optima.

  Crossover probability:

- Too High: A high crossover rate it might lead to a chaotic search process where good solutions are broken apart as quickly as they are formed, making it difficult to converge to an optimal set of compiler flags.

- Too Low: On the other hand, one too low would not allow sufficient mixing of genetic information, reducing the algorithms ability to combine the better flags to form “fitter” individuals. This would result in a slower exploration of the solution space, where there is a possibility of getting stuck in a local optima.

  Therefore, the key of genetic algorithms would be balancing these probabilities to ensure that the search process is not too random or constrained. An optimal balance would allow for the exploration of a broad solution space effectively converging to a best solution.

Local search

“local search algorithms are useful for solving pure optimisation problems, in which the aim is to find the best state according to an objective function.” [11]

Local search algorithms are powerful tools for searching vast and complex search spaces, through a methodical approach local searches can navigate towards optimal solutions. Local searchers operate using a single current node and move only to neighbours of that node [10]

Local searches can take on heuristic functions which estimate “closeness” to a goal or the quality of a solution, alternatively we could guide our local search by applying a objective function on each node where the focus is on minimising the execution time.

Hill climbing

In hill climbing, the aim is to optimise a certain objective function, starting at a initial node, the algorithm aims to reach a peak iteratively. Each step in the hill climbing process involves evaluating the neighbourhood of the current solution. In general, hill climbing algorithms are known for their efficiency, requiring only a limited amount of resources; some memory to store the current state and limited processing power to identify and evaluate neighbours. [12]

Hill climbing’s strategic evolution requires a method of generating neighbouring nodes from its current node, and have a testing methodology to judge them. Unlike other search algorithms, hill climbing lacks a backtracking feature, therefore eliminating any possibility of revisiting or reevaluating past decisions. This lack of a design feature and no introduction of diversity, often leads the algorithm to get stuck in a local optima, settling for local maxima's rather than identifying the global maxima. Another consequence is getting stuck on a plateau, which is a flat area of the state-space landscape where no close better solutions exist. [10]

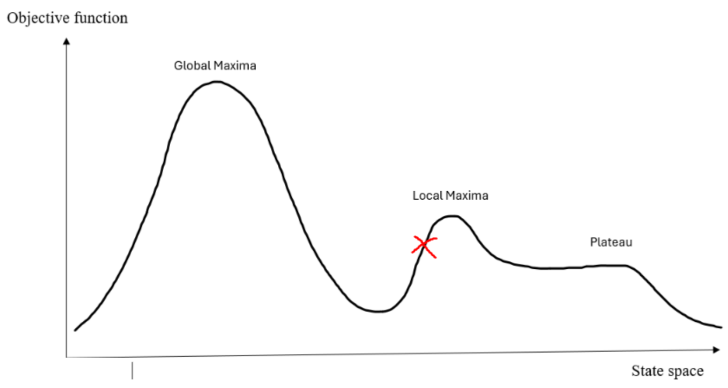


Figure 2: Objective function – state space graph depicting the local and global points and how they could misguide our algorithm

In the above figure, a visual representation depicts the issue such local searches face, the current state, denoted with the red cross, would tend towards the local maxima, rather than take a performance hit but achieve the global maxima. When reaching a plateau since no better solutions exist, the algorithm would be stuck.

Hill climbing presents different types, each stemming from a foundational structure and adhering to the fundamental principle of incremental improvement and optimisation

1) Start with initial node
2) Check if the initial node satisfies goal conditions
3) Enter loop to search for a better state
4) (here the logic differs depending on the implementation used)
5) End the process if no improvement is found

Simple hill climbing

This variant evaluates neighbouring solutions and adopts a "greedy" approach where it selects the first one that improves the value of the objective function, regardless of whether it's the best possible improvement amongst all neighbours.

Since simple hill climbing doesn’t necessarily examine all neighbours before making the move, this makes it faster but potentially less likely to find the best solution.

However, as mentioned above this approach is likely to get stuck in a local maxima or plateau. The algorithm is defined as shown:

```
Function simple-hill-climbing(problem):
current_node <- initial_node
while (current_node is not improved) do
Neighbourhood <- get_neighbours(current_node)
for each neighbour in neighbourhood do
    If neighbour.value > current_state.value then
        current_node = neighbour
        break
end
return current_node
```

This pseudo code outliens a simple hill climbing algorithm, the algorithm starts with an initial node, the choice of the initial node can be arbitrary or based on some heuristics related to the problem.

The algorithm then enters a while loop that continues as long as there is an improvement made from the current node, the termination ensures that the algorithm does not enter an infinite loop

At each iteration, the algorithm generates a neighbourhood of the current node, this neighbourhood consists of neighbours which are slight variations to the current node.

For each neighbour, the algorithm evaluates it using some function, if the neighbour is found to have a better value then the algorithm updates the ‘current_node’ to this neighbour. The loop breaks as soon as a better neighbour is found. [12]

Steepest ascent hill climbing

A small but drastic tweak we can make to simple hill climbing is to explore all neighbours before choosing the optimal neighbouring solution, rather than selecting the first improvement encountered relative to the current state.

Similar to simple hill climbing, this variant also can get stuck in local maximas or plateus, however the thoroughness of evaluating all options within a neighbourhood increase the likelihood of producing solution closer to the global maxima, albeit at the expense of far greater execution time. [13]

```
Function steepest-hill-climbing(problem):
current_node <- initial_node
while (current_node is not improved) do
Neighbourhood <- get_neighbours(current_node)
Best_value <- current_node.value
for each neighbour in neighbourhood do
   If neighbour.value > current_node.value then
          current_node = neighbour
        best_value = neighbour.value
end
return current_node
```

This code is almost completely similar to that of simple hill climbing, however we introduce a boolean variable, best_value, to keep track of the optimal solution encountered thus far, this allows the comparison of neighbouring solutions with the best one identified.

Stochastic hill climbing

Unlike it’s deterministic counterparts, stochastic hill climbing introduces randomness into the selection process of the next move, this randomness is not completely unconstrained; the algorithm will be bias towards neighbours that offer improvements over the current state, known as “uphill moves”. Choosing an uphill move at random, rather than always selecting the best uphill move (steepest) introduces variability whilst also ensuring the algorithm converges to a perfect solution. [14]

This introduces a number of advantages over our traditional deterministic counterparts:

- Escape from local maxima: Randomly selecting among uphill move provides a mechanism for the algorithm to potentially escape local maximas by allowing it to take less optimal sets in the short term in hopes of discovering better solutions long term, this follows on to our next point

- Exploration vs exploitation: In optimisation, exploration refers to searching through the solution space to discover different solutions, whereas exploitation involves making use of known good solutions to find even better ones, the random selection among uphill moves strikes a balance between the two

Each hill climbing technique offers a unique approach to optimising a solution and will inevitably converge to a maxima, whether that be a local maxima or global maxima is something uncertain. Each variant has it's own advantages and drawbacks. For instance, while simple hill climbing is resource-efficient and straightforward, it is more susceptible to getting trapped in local maxima due to its greedy nature. Steepest ascent hill climbing, by methodically evaluating all neighbours before moving, increases the chances of finding a better overall solution but at the cost of higher computational effort. Stochastic hill climbing introduces randomness into the selection process, which helps in escaping local optima but does not guarantee the most efficient path to the global maximum.

In contrast, genetic algorithms leverage a broader set of operations inspired by natural selection, this approach offers far more variability and a better balance between exploration and exploitation. As a result, in mapping problems it was shown to significantly outperform simple hill climbing. [15]

Variable Neighbourhood search

Variable neighbourhood search is an effective metaheuristic method for solving a set of combinatorial optimisation and global optimisation problems [16]. It explores distant neighbourhoods of the current solution and moves from there to a new one if and only if an improvement is made. This systematic changing of neighbourhoods is combined with a local search to explore each individual neighbourhood.

The core functionality of VNS involves three main steps: shaking, local search and moving.

The shaking phase alters the current solution to generate a candidate solution in a different neighbourhood, this is for introducing diversity. After this a local search is run on the new solution to find better neighbouring solutions. If an improvement is observed, the algorithm adopts this new solution and the cycle repeats. If not, the algorithm expands the search radius by increasing the neighbourhood size and proceeds with another iteration.

VNS' most significant attribute is its variability, the shake phase, similar to the mutation probability of genetic algorithms, it adds diversity to the state allowing for comprehensive exploring of new areas of the solution space, whereas the local search is able to intricately explore small search spaces. The balance of exploring new areas of the solution space while intensively searching within promising regions, not only overcomes the inherent limitations of local searches in getting stuck in local optima but makes VNS a great solution when tasked with optimisation problems.

```
While stopping condition is not met do
  K <- 1
  While k <= k max do
    S' <- shake(s, k)
    S'' <- localsaerch(S')
    If (S'' is better than S):
            S <- S''; k <- 1
    Else
            K<- k + 1;
  End
End
```

The pseudocode above defines the variable neighbourhood search, it starts off with initialising the neighbourhood size counter, k, to 1. In VNS, k represents the size of the neighbourhood being explored, starting with a small neighbourhood ensures that the algorithm first searches close to the current solutions.

This loop continues as long as the current neighbourhood size, k, is less than or equal to the maximum neighbourhood size, or until k reaches a certain number. Kmax determines how far the algorithm will search, this loop systematically increases the neighbourhood size and explore solutions that are progressively further away from current solution.

The shake step generates a new solution, S', by perturbing the current solution. The importance of the shake can not be understated due to the varaibilty that it introduces, making VNS an algorithm suitable for large search spaces

After shaking, a local search is performed starting from the new solution S', to find a local optima, the local search will search neighbouring solutions of S', the variant of local search used can vary , however hill-climbing is a typically coupled with VNS [16].

Lastly the conditional checks if S'' prime is better than S according to some cost function, if this is the case then S'' prime is assigned as the current state (new best state) and k is set to 1 again, ensuring that the algorithm again starts searching close to this new solution before gradually expanding the search radius. Otherwise, the neighbourhood size is incremented, this shift to a larger neighbourhood is based on the principle that a local optimum with respect to one neighbourhood structure might not be a local optimum with respect to another, allowing the search to continue and potentially escape from local optima. [17]

This whole process is looped until the stopping condition is met.

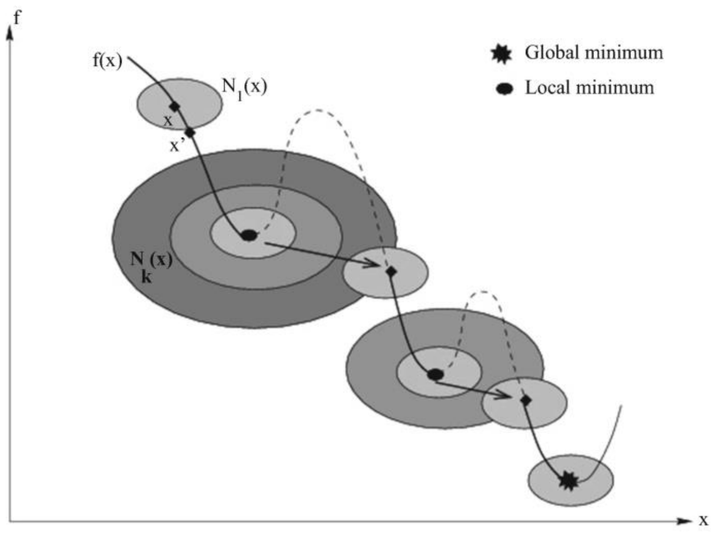


Figure 3: objective function – state space graph depicting how VNS can venture out of local minima by its neighbourhood expansion shaking. [18]

Here is a visual representation of the progress VNS makes.

The point x represents a candidate solution in the state space, the encompassing circle represents the neighbourhoods around the solution, N1(x) is the immediate neighbourhood, and as k increases, Nk(s) represents progressively larger neighbourhoods. At the start of the f(x) curve, we can see the candidate solution make progress to x' with k=1, however once we reach a local minima, we require k to increase, allowing us to search broader neighbourhoods, this then ,as noted by the arrow, allow us to escape the local minima. This continues until we reach the global minima, or the condition in the while loop is satisfied.

A local search algorithm as discussed would return a solution of the first local minimum, the shaking mechanism of VNS allow us to circumvent this.

Variable Neighbourhood Search Variants

Since VNS was first proposed in 1997 and has a lot of flexibility, it has rapidly developed to have many variations all suited for slight different purposes.

Reduced Variable Neighbourhood search (RVNS) is a streamlined version where the search skips the local search phase and only evaluates new points found randomly within the neighbourhoods, these values of the new points are compared with the current state and updated if an improvement is found. RVNS's method of jumping to random points within the neighbourhood and evaluating them against the incumbent avoids the process of local search and instead relies on random sampling to find improvements. This is useful for problems where local search is expensive especially in very large instances.

Basic Variable Neighbourhood search combines deterministic and stochastic elements, it introduces randomness through the shaking phase followed by a deterministic search. This variant is as discussed above.

Skewed Variable Neighbourhood Search (SVNS) addresses the problem of exploring valleys far from the incumbent solution [17]. It does so by permitting transitions to less optimal solutions with a certain likelihood based on a skewing function that takes into account the “distance” from the incumbent , thus offering a counterbalance to the common issue of becoming stuck in a local optima. The distance here would be as defined in terms of solution space, in our case it would be the number of differing flags from the incumbent state. The skewing function adjusts the acceptability of a new solution based on how different it is from the incumbent solution, a parameter, denoted by an alpha, determines the degree of skewness, this would determine whether to accept worse solutions in hopes of escaping local optima’s. A higher alpha would make the algorithm more adventurous. SVNS is particularly useful when the search space is vast and complex, with many local optima scattered throughout.

Machine learning – Neural networks or deep reinforcement learning

Machine learning, more specifically deep reinforcement learning, offers a entirely different approach to the problem of compiler optimisation. Rather than experimenting with different compiler flag combinations as we have seen with all the strategies thus far, machine leaning goes down the avenue of static analysis, which assess the code without executing it. By examining the intermediate representation of the source code, machine learning models can predict the impact of various optimisation strategies without the need for program execution. Encoding what each compiler flag does , means our algorithm could hand select the based compiler flags based on what it observers in the source code.

Alternatively, a more practical approach would be to train a DRL agent to dynamically select and order compiler optimisation passes. These passes include the sequences constituting LLVM’s O3 optimisations, which the agent learns to outperform. This is accomplished without relying on dynamic features that require runtime information. Instead, the agent relies solely on static information available at compile time. [20]

In this framework, we must model our optimisation problem as a reinforcement learning problem. The first step is to use static analysis to extract features from the code, these might be number of instructions, control flow graph complexity, types of operations (arithmetic, logical, memory access), and potential bottlenecks.

Through the learning phase , the goal would be to train with deep reinforcement learning (DRL) ,acting as the DRL agent to learn to associate static analysis features of the program with the cumulative rewards to applying certain compiler optimisation. The agent would experiment with different sets of optimisation flags, observes the resulting changes in program performance, and updates its internal model based on the rewards received. [21]

As opposed to the aforementioned algorithms we have seen, this approach will require a lot of sanitised data. For supervised learning, the training dataset would require labels indicating the best optimisation flags used, these would be determined through empirical testing.

For reinforcement learning, the labels are discovered through interaction with the environment. In this case, the environment is the compiler system and the reinforcement learning agent would learn by applying optimisations and observing their effect on program performance. [20]

This unorthodox approach has numerous advantages and disadvantages.

1. Extensive training data required: to effectively train our model and generalise across a wide range of programs, the models requires a vast, diverse and rich training dataset. Sanitizing and gathering this data can be a substantial task, but also does not guarantee that each dataset would cover all wide array of programs available
2. Training a deep reinforcement learning model on a large dataset is computationally expensive and time-consuming, rather than a simple algorithm that requires little computation power and can be run on any machine, using machine learning would not allow much portability to other machines.
3. The black box nature can complicate debugging and refining the model.
4. There is a potential for overfitting and underfitting, there is a risk that the model becomes overfitted to the types of programs and optimisation scenarios seen during training, leading to less effective optimisations for new or different types of programs.

On the other hand,

1. Unlike traditional autotuning approaches that require multiple runs of the program with different flag combinations, a DRL-based model can predict effective optimisations before the program is compiled and run, saving time and resources
2. The model can continuously update and refine it’s understanding of effective optimisations as it is exposed to more programs, this leads to optimisation strategies constantly being able to improve.

Using machine learning is an innovative approach and a shift from traditional optimisation techniques. This approach utilises the power of static analysis and would behave in the most “human-like” way. Although the efficiency of a trained model is unmatched, the process of collecting and sanitising not only a vast array of data but a rich and diverse one too is a substantial task.

Comparison and effectiveness of strategies

All four distinct strategies- genetic algorithms, local search, variable neighbourhood search, machine learning – have been used for compiler optimisation. Among these, genetic algorithms are the most widely used strategy for tackling optimisation problems for compilers. By treating each individual flag as a gene, the use of genetic algorithms as a logical and well-tested approach when trying to emulate the process of natural selection. This approach not only is supported by multiple prior studies where it's efficacy is demonstrated, but is also validated by the principle of the survival of the fitness as observed in nature. [22]

All papers concluded that genetic algorithms to be a suitable candidate to determine the compiler flags due to the large search space, [23]

The effectiveness of this method is shown where it outperforms -O3 in almost all scenarios, in some cases significantly improving runtime, with some programs showing up to a 40% decrease in runtime when compared to using the compiler's -O3 optimisation level. [24] [25] [26]

Using local search to probe this issue is not well researched yet, however this is mostly due to the theoretical understanding and inherent flaws that immediately indicate that this strategy would not ideal in an optimisation problem, which not only has a large search space but a lot of potential local optima's and plateaus.

There has been some work where an impatient hill-climber has been used in the context of adaptive compiler optimisation, focusing on the application of scalar optimisations to low-level intermediate code. [27]

Impatient hill climbing is a variant of simple hill climbing that introduces a limit on how long how the algorithm will search in the neighbourhood for a solution before restarting, if the algorithm does not find a solution within some defined metric (i.e time constraint) then the algorithm becomes "impatient" and restarts with a new random solution, rather than exhaustively searching all neighbours. This algorithm aims to find local minima efficiently, and then utilises loop unrollers to discover more effective optimisation sequences, since loop unrolling can substantially improve the performance of code that relies heavily on loops, it’s inclusion mean that the optimisation algorithm has a greater variety of transformations to choose from.

Despite the effectiveness of this adaptive compiler showing a 25% performance increase the algorithm's tendency to get stuck in local minima makes it less promising. The approach diverges significantly beyond this point through the use of loop unrolling, while our focus remains on exploring different combinations of flags. Therefore, the local search in it’s basic form would not be suitable for our problem. [27]

There has been no attempt to use machine learning techniques to find hotspots in code and then choose the best optimisation flags based. Utilisation of machine learning for detecting lithographic hotspots in integrated circuit layouts, represents a shift from the traditional mtethods of pattern matching to more advanced, data-driven approaches. In our paradigm, hotpsots would refer to multiple loops, etc

The initial methods for hotspot detection relied heavily on pattern matching (PM), which was effective to known hotspot patterns however failed to identify never-seen-before patterns/ Since source code can be written in different styles (types of loops used, loops in different functions adding up, etc) this limation may be even greater in our case. [28]

However, machine learning, especially supervised methods emerge as promising alternatives in finding code hotspots, this study reveals that by compiling a more realistic dataset, containing a mixture of code smells and getting the right balance between smelly and non-smelly instances, the effectiveness of machine learning has nearly solved the problem of code smell detection. Despite this, the conclusion was that “more research is needed towards structuring datasets appropriately and jointly with the predictors to be used”. [29]

In our case significant challenge to this approach would be dataset composition, model generalizability, defining what hotspots are, and linking the effects of optimisation flags to hotspots.

Variable neighbourhood search adeptly addresses the issue of local search getting stuck in local optima, by introducing neighbourhood changes and random shaking. There has been limited research on VNS for compiler optimisation using AI, however it has been used extensively to solve optimisation problems with great success.

VNS has been used for solving the traveling salesman problems, used as a heuristic for large scale vehicle routing problems and for continuous optimisation. VNS is particularly useful for problems where the search space is complex or poorly understood. [30]

ParamILS

ParamILS is an automatic framwork designed for optimising the performance of algorithms, it uses iterated local search for parameter tuning and algorithm configuration. The aim, similar to ours, is to automatically identify settings that lead to the best algorithmic performance.

At its core, ParamILS employs a local search based approach, the iterated local search enhances simple local search by incorporating mechanisms to escape local optima, such as solution perturbation and an acceptance criterion for new solutions. This allows the exploration of the configuration space beyond local neighborhoods.

The effectiveness of ParamILs has been shown across various contexts, from configuration commercial optimistaion tools like CPLEX to optimistion algorithms for SAT solving and other NP-hard problems. ParmaILS has consitennly found configurations that improve performance.

This approach that is grounded with ils works as follows:

1. Initialistaiton: it starts with an initial state configuration, this can be based on defaults or chosen at random
2. Local search: the method iteratively explores neighbouring configurations, where neighbours differ in the setting of one parameter, to find improvements in algorithm performance
3. To escape local optima, paramILS introduces changes to the state by perturbating it, to move the search to a different region, it periodically makes larger, more disruptive changes to the state until it detect that it escapes the local optima.
4. Acceptance criterion: it decides whether to accept or reject the new configuration based on its performance compared ot the current configuration
5. Adapative capping; this technique limits the evaluation time for configurations that appear unlikely to outperform the current best, saving computational resources.

In the context of flag searching, parmaILS offers a systematic and automated approach to explore the combinatorial space of possible flag settings, its unique usage of local search coupled with solution perturbation provides a promising basis to create a similar framework when dealing with our problem. [31]

Requirements and specification

Functional requirements

FR1: Computation resources

- Multi-core processors are required for parallel processing, this is necessary for implementation of parallel variable neighbourhood search strategies to work efficiently
- Access to sufficient memory to handle multiple compilation and program executions

FR2: Software requirements

- Development tools and libraries for Python 3.11 and C/C++ are necessary for implementing algorithms and benchmarks programs, packages must be installed with the requirements.py

FR3: Algorithm implementation

- Develop a system capable of identifying an optimal set of compiler flags that minimise execution time for various C/C++ programs.
- Optimise and implement parallel variable neighbourhood search (PVNS) and hill climbing (HC) algorithms to explore solution space efficiently.

FR4: Parallel Processing Utilisation

- Implement parallel processing techniques to expedite the search process and find better solutions if available.

FR5: Testing and evaluation

- Gather a diverse collection of C/C++ programs that will be used as benchmarks to test the flag optimisation's efficiency.
- Define clear performance evaluation metrics focusing on gathering the lowest execution time possible. Additional metrics might include memory usage, compilation time, and other relevant performance indicators.

Non-functional requirements

NFR1: Usability

- The system should provide the user with a user-friendly CLI interface for inputting problems and allowing customisation of algorithm from the CLI

NFR2: Scalability

- The system must efficiently handle an increasing number of compiler flags and different types of benchmark programs without a significant drop in performance

Environment requirements

The system's performance in optimising compiler flags is slightly dependent upon the execution environment, including the hardware architecture and available resources (CPU cores, memory). Parallelisation can significantly influence the outcome by reducing execution times, however if the system environment does not meet specific requirements, parallelisation could have a determinant effect, as each core would be shared during runtime of multiple processes.

Another important issue that arises with the environment is noise. When running some programs that take seconds to run, noise can significantly affect that by +- 0.5 seconds, which could not only produce incorrect results, but more importantly misguide the algorithm potentially ruining the whole search.

To address this, we have two measures in place, firstly our search will be run on an Amazon AWS EC2 instance, specifically a t2.medium, which contains 2 vCPUs and 8GB of RAM. Having a clean instance with no background programs but only extremely minimal and basic linux threads running allow us to achieve a system with little to no interfering noise, this will help keep our results consistent and not misguide our search. On top of this having a simple build of ubuntu is far less stressful on our running program than Windows 11 with Windows Subsystem for Linux or MAC OS.

The second measure in place will be discussed in the design of our algorithm.

Design

Overview

Given the insights gained from exploring various strategies it would be ambitious but potentially rewarding to implement a variable neighbourhood search augmented with hill climbing. This approach promises to leverage the strengths of both techniques, potentially offering a more effective solution in navigating the vast and complex landscape of compiler optimisation flags.

Genetic algorithms are the most commonly employed method for addressing this particular kind of problem, making it the safest approach. However, to introduce some innovation inspired by the successful strategies that incorporate local search such as ParamILS as well as research that utilised impatient hill climbing, I chose to employ local search, with VNS to avoid local minima, the shaking process of variable neighbourhood search (VNS) mimics the mutation probability of the genetic algorithms, so despite not opting for the conventional choice of genetic algorithms, our current strategy closely resembles with their foundational principles. .

Referring back to our problem, the GCC/G++ and clang-based compilers provide us with 174 optimisation flags, these flags cover a wide range of optimisations such as assuming strict aliasing rules, re-ordering functions in order to improve code locality and performing variable operations on trees. These vast array of flags are not applicable to every program, and already well -optimised programs may not be affected at all by such flags. Our objective is to form a conglomerate of these flags such that after compilation they provide a faster execution time than that of the -O3 preset.

Variable Neighbourhood Search (VNS)

Since we are using an effective trajectory-based metaheuristic method with a fast convergence for solving a set of combinatorial optimisation problems and global optimisation problems, we will remodel our problem into a combinatorial one.

Given a program P, and a set of flags F = {f1 , .. fn}, our problem consists of establishing a binary vector B = (b1, .. bn) where b is a part of the binary set {0,1}, which minimises the execution time of P when its compiled with the set of flags indicated by the binary vector.

The binary vector takes two values, 0 or 1. 0 denotes the flags being de-activated , whereas 1 denotes the flag being active.

We will define our fitness function as the execution time of a program, the goal for would be to minimise the value outputted by the fitness function.

Now that the flags are encoded into binary strings, we will have 174-bit long strings, where each one defines a certain state or node, a neighbour of a node would be all bit strings that are 1-Hamming distance away. For example, the 1-Hamming distance neighbours of 0100 would be 1100, 0110, 0101 and 0000.

The choice of encoding flags as bits as several advantages, most notably efficient operations: Bitwise operations are among the most efficient operations in computer architecture, flipping a bit to enable or disable flag is a simple operation, additional representing each flag as a single bit significantly reduces memory footprint compared to other encoding methods that might require strings

Basic Variable Neighbourhood search

Our approach will be using basic variable neighbourhood search coupled with a hill-climbing algorithm, the effectiveness of both algorithms and it's balance with exploring both the breadth and depth of the search space makes it a suitable solution for our defined problem.

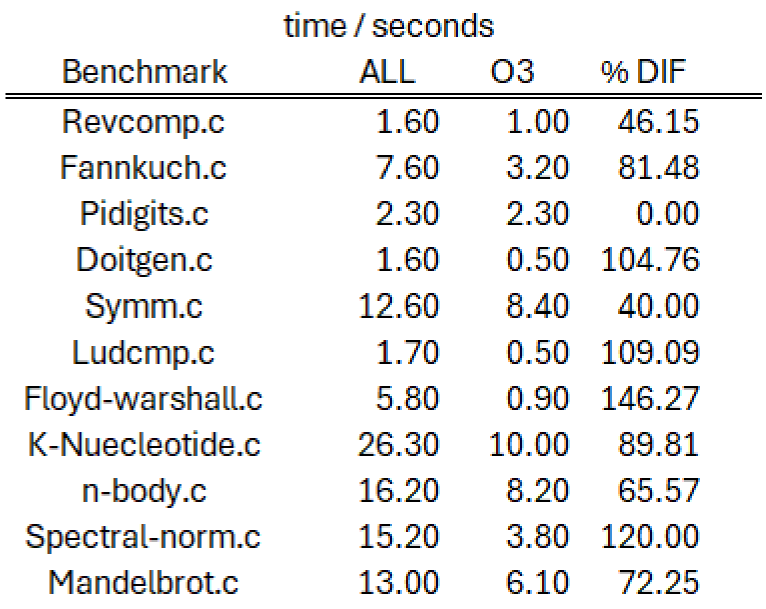
Basic variable neighbourhood search follows the basic idea of:

- Shaking – perturbs the bit string by k to introduce diversity into the search process
- Local search – use a searching algorithm to explore all neighbours, which are defined as 1-Hamming distance away
- Neighbourhood change – if a better solution is found by a shake or local search, update the neighbour and restart

To initiate this algorithm we must start with an initial state, there are four approaches we have for determining an appropriate starting point

- Random Starting State: We initiate our search with a 174-bit long string, randomly altering it to ensure we commence from a diverse starting point.
- Start with no flags and build up – we start with no flags enabled and then systematically incorporate them into our configuration. This method would yield initial performance enhancements due to the gradual build-up of optimisation layers.
- Adapting from -O3 – given that -O3 has been determined to be the fastest default flag, we can bypass the initial slow phase by adjusting our 174-bit string starting from the -O3 configuration. This provides a quicker entry into optimisation
- Iterative flag removal - since our the idea of optimising mainly depends on removing unnecessary flags to eliminate overhead this approach where we discard superfluous flags acts as a top-down approach, where we methodically remove flags, streamlining towards constant refinement.

The opportunity to leverage prior knowledge or insights above effective flag combinations is overlooked by first two approaches. According to the documentation the -O3 flag turns on most optimisations which are 111 flags in total, however after experimenting by activating all the 111 specified flags manually without the -O3 flag enabled, we see that -O3 massively outperforms it in all programs.



	time / seconds		
Benchmark	ALL	O3	% DIF
Revcomp.c	1.60	1.00	46.15
Fannkuch.c	7.60	3.20	81.48
Pidigits.c	2.30	2.30	0.00
Doitgen.c	1.60	0.50	104.76
Symm.c	12.60	8.40	40.00
Ludcmp.c	1.70	0.50	109.09
Floyd-warshall.c	5.80	0.90	146.27
K-Nuecleotide.c	26.30	10.00	89.81
n-body.c	16.20	8.20	65.57
Spectral-norm.c	15.20	3.80	120.00
Mandelbrot.c	13.00	6.10	72.25

Figure 4: Performance table showing the various benchmarks against the ALL and O3 configuration

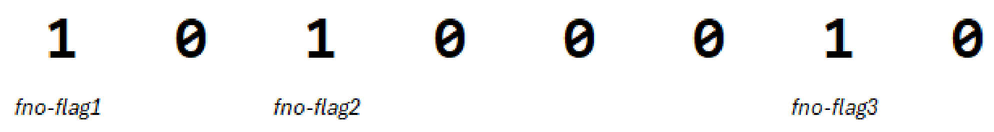
We observe that on average O3 offers a nearly 80% performance increase relative to enabling all O3 flags manually. This indicates that O3 is doing something more than just enabling the flags stated in the documentation.

Due to this instead of assigning binary values to each of the 111 O3 flags to indicate their activation (1 for active, 0 for inactive), a more effective strategy would be to invert this logic. Now, a flag set to 1 indicates deactivation whereas 0 indicates that the flag remain active within the O3 optimisation level. The additional 63 flags that are not activated by O3 will follow the old logic. These flags are typically more risk-adverse and cautious in nature hence having a starting point with all 174 flags enabled would lead to compilation errors in numerous programs.

To enable a flag with the GCC/G++ command we use the -f tag followed by the flag we want to enable, i.e -fmerge-all-constants

To disable a flag we use the -fno tag followed by the flag we want to disable, i.e -fno-ira-share-spill-slots

In our case when running GCC/G++ with the -O3 tag we will mainly be using the -fno tag to disable flags.



As shown above the bitstring correlates to the following command

gcc -O3 program_name.c ***-fno-flag1 -fno-flag2 -fno-flag3*** -o benchmark_program

As already defined, our fitness function is the execution time of the compiled program with the program and flag set. Measuring the execution time accurately poses a challenge due to the internal process of the computer producing noise. We have already introduced AWS instances to eliminate most noise, but as an extra precaution to deal with this issue we execute each program 3 times and pick the minimum value. The minimum value is used since it produces the execution time that has the least amount of external interference.

Since our state is encoded as bit strings, this enables us to apply traditional variation operations. Shaking refers to a bit-flip of our bit string, we begin by altering a single bit in the solution (k=1) to identify a neighbour. Upon detecting convergence – when we explore n neighbours and none of them yield a improvement – we increase our k value, so we end up modifying two bits, this incremental increase continues until we reach a certain condition.

When we improvement is found, the k value is put back to one, so we can explore neighbours closer to the new best solution.

Shaking and bitstrings

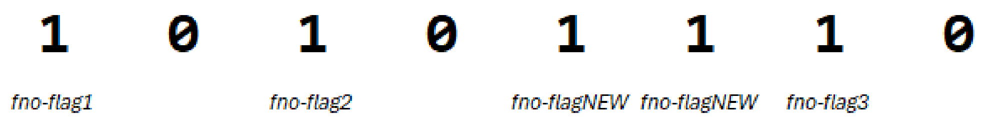
Here we have initial state as defined

1	**0**	**1**	**0**	**0**	**0**	**1**	**0**
fno-flag1		*fno-flag2*				*fno-flag3*	

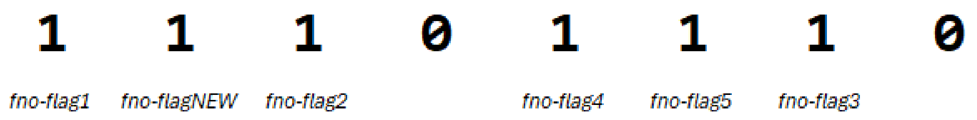
We will first shake our state where **k=1**, this randomly flips a bit , which in turn disables a flag

1	**0**	**1**	**0**	**1**	**0**	**1**	**0**
fno-flag1		*fno-flag2*		*fno-flagNEW*		*fno-flag3*	

The local search algorithm is run and no improvement is found, so the algorithm increases the k+=1 to 2 (**k=2**) in this case this will randomly flip 2 bits, which in turn disables two flags



After conducting a local search from the new state, we find a state that is better than the best solution found thus far, now the k value is reset back to 1 (**k=1**) and the next shake will only bit-flip one bit so we can explore closer to the best solution.



Local search

Our next stage of the algorithm is local search. Many different types of local search have been discussed, especially hill climbing variants. The large number of flags introduces a practical challenge; specifically, each state would be surrounded by 173 neighbours. Utilising a steepest ascent hill climbing strategy, which involves evaluating every neighbour to identify an improvement, would significantly slow down the process. For algorithms that have a 10 second execution time it would take nearly 30 minutes just to evaluate one neighbourhood. This is not a practical solution although in theory it would provide the best results.

Simple hill climbing, which selects the immediate best neighbour it encounters, would provide a more feasible approach. Substantially reducing the search time, this methods strikes a balance between efficiency and effectiveness.

However, when a local optima solution is found, this algorithm would face the same fate as steepest-ascent hill climbing where it would have to endure one complete neighbourhood search without yielding an improvement, although this based on the likelihood of getting good solutions and not a design idea of the algorithm, long execution times would still be something to expect when running with this approach.

Impatient hill climbing would be redundant in our setup, since our VNS algorithm inherently guarantees not getting stuck in local optima thanks to our shaking mechanism, the quick abandonment of search paths that don't immediately yield improvements would slightly nullify the impact of shake.

The same applies to stochastic hill climbing that introduces randomness for uphill actions.

Since, simple and steepest hill climbing both face the issue of long execution times, we've opted to impose a cap on the number of neighbours we explore to ten.

To chose the better algorithm, we evaluated their performance. For this we used two programs reactive to flag changes and plotted the results.

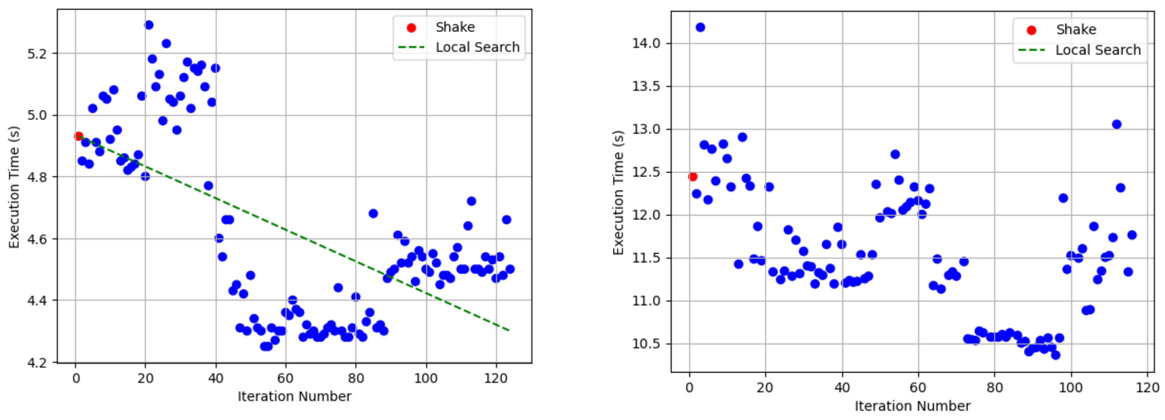


Figure 5.1: Simple Hill Climbing of two benchmark programs: Fannkuch.c and k-nucleotide.c

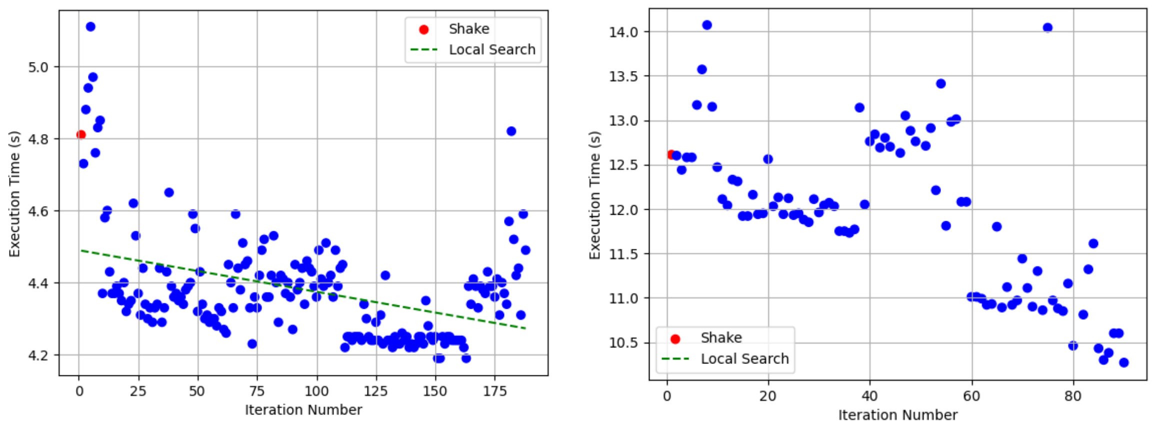


Figure 5.2: Steepest Hill Climbing of two benchmark programs: Fannkuch.c and k-nucleotide.c

The red dot indicates the starting state which is -O3 enabled, and the blue indicates each local search of a neighbourhood. In comparing the effectiveness of simple hill climbing and steepest hill climbing, the graphs indicate that the steepest hill ascent demonstrates a more aggressive approach towards finding improvements, however it take significantly longer. Despite simple hill climbing taking more iterations to reach the same improvement, it still does so in less time. Therefore, the decision to favour simple hill climbing over steepest hill climbing is thus grounded in the prioritization of time efficiency.

Our simple hill climbing algorithm will work by taking 10 neighbours, where a neighbour is defined as being 1-Hamming distance away from the current node. If an improvement is found within those 10 neighbours, the algorithm will transition to that improved state. However, if no improvement is found, the local search concludes without any changes, rather we loop back to the start of the code, initiating another round of shaking again. Research shows that hill climber finds good sequences in five to ten descents.

Therefore this limit is appropriate and not overly restrictive, ensuring we don't consistently overlook better solutions.

Despite variable neighbourhood search solving the issue that local search faces of getting stuck in local optima, it still relies on predefined neighbourhood structures to explore the solution space, the algorithm might miss better solutions outside its exploration space. For instance, the initial steps of the algorithm may uncover an improved solution compared to the starting point, however this doesn't necessarily mean it's the best possible solution, even if it appears optimal from that specific juncture onward.

Parallel VNS

Parallel VNS instance can explore various parts of the solution space simultaneously, increasing the likelihood of uncovering multiple local optima, this additional diversity helps mitigate the risk of our algorithm getting trapped in a sub-optimal region, and increases computational efficiency but leveraging multiple cores, significantly speeding up the search process. [19] .

There are several ways of parallelising VNS, in our case, the most appropriate options would be to:

(i) Initiate multiple homogenous VNS processes concurrently – inspired by ParamILS, each parallel process or thread performs a search independently. Due to the random nature of shaking, all independent processes are likely to have different starting paths to a solution. This will aim to explore more of the search space, although this approach does not speed up the process it will introduce more variability.

(ii) Farmer worker model - Each parallel process or thread takes a single neighbour and performs a local search independently. The parallel processes do not start from different initial solutions but rather explore different paths from the same starting point. A local search is performed on one neighbour at a time, but multiple such searches are conducted in parallel. In light of the name of the mechanism, it can be seen as a farmer deploying workers to find the best solution at a given point. VNS instances regularly exchange their current solutions throughout the execution, creating a cooperative environment that enhances the overall search process.

Our complete framework has now been developed including choosing a suitable starting location, and including parallelisation.

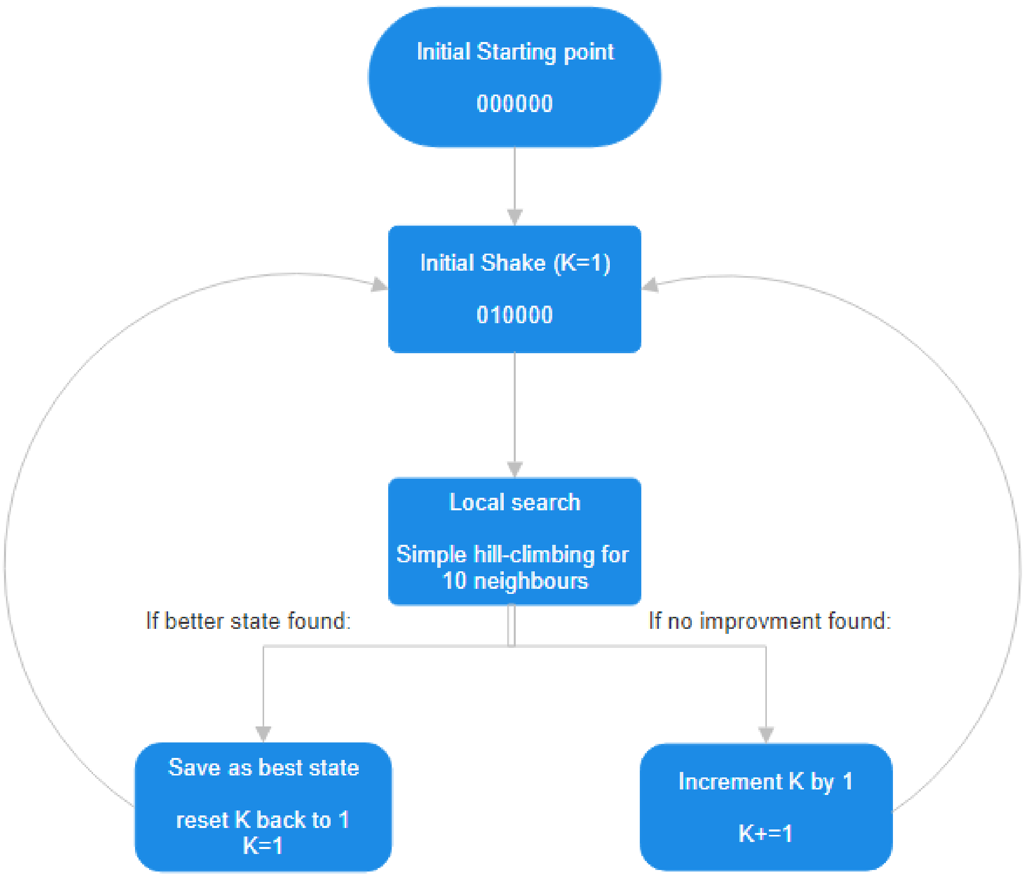


Figure 6: Flow state chart depicting the core functionality of VNS

A flow diagram of the internal mechanism of our algorithm

As described this is the flow of our algorithm, we begin with an initial state and then go into a local search, the 10 neighbours will be chosen at random from the complete set of all neighbours, if a better state is found from our local search, then we will narrow our algorithm to focus on that specific area, otherwise we will increment k by 1 to can expand our neighbourhood search area.

This is all encapsulated in a while loop until a certain stopping condition is fulfilled.

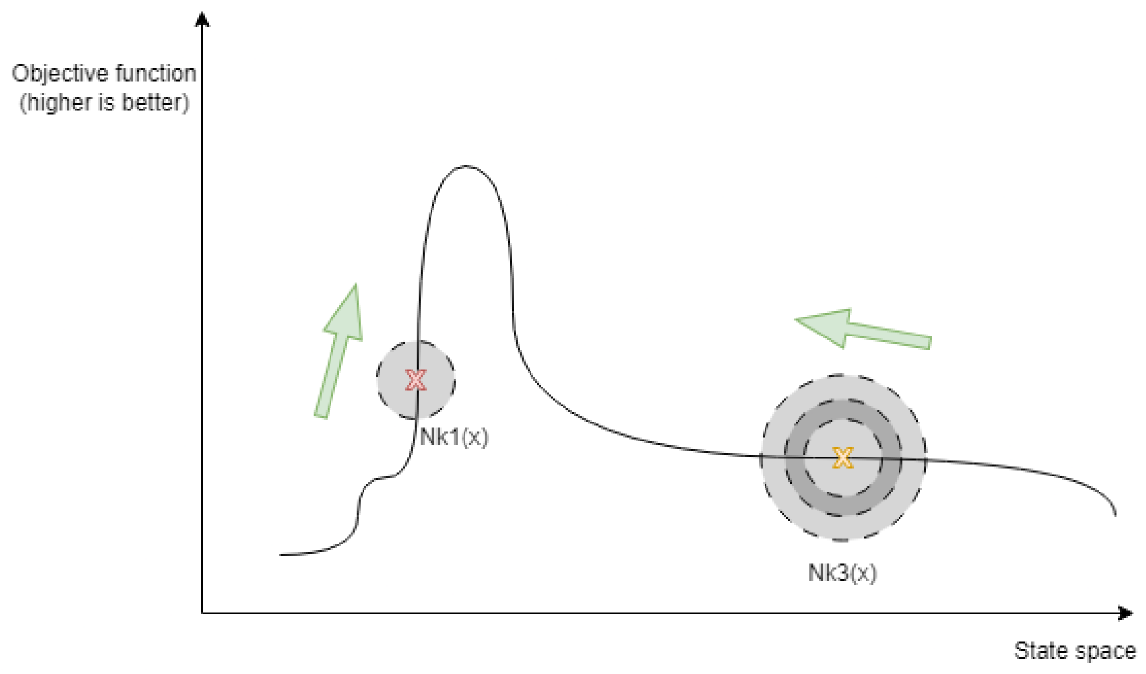


Figure 7: Objective function – State space graph depicting how different state behave based on their position within the state space

Upon running our algorithm we should observe an convergence to local optima as shown above. Here, NKn represents the neighbours around a state of degree K, where K is level of neighbourhood being explored, it defines the size of neighbourhood around the current solution that the algorithm is considering for potential improvement.

As seen the red state only requires k to be 1, since it is able to find a new best improvement immediately within its neighbourhood. In contrast, the yellow state requires are much greater value for k since it’s situated on a flat space where improvements are sparse. The shaking mechanism provides this ability, whereas a plain local would not making any significant movement.

This diagram also explains how parallel processing, specially initiating multiple homogenous VNS processes, may benefit our program. Both states shown on the diagram would represent a separate core, One core might reach a solution more quickly than another, although in our case, both states will eventually converge to the same local optima. However in our state space, this might not always be the case; one process could discover a superior local optimum compared to another. Employing multiple processes would thus increases the probability of our algorithm returning a global optimum in every run.

Benchmarking

In order to obtain a comparable, representative and extensive set of programs we have used The Computer Language Benchmarks Game (CLB) and PolyBench/C to select a mix of the benchmarking programs. [32] [33]

The benchmark programs vary in description but also execution time, we have collected a sample of 14 benchmark programs, 8 from the CLB and 8 from PolyBench. These programs cover a wide range of computational tasks, including data mining, solving algorithms, analysing "k-nucleotide" sequences within DNA or RNA strands, generating and writing random DNA sequences, hash table operations, and more. Incorporating a variety ensures that our testing environment can accurately reflect the diverse challenges and performance needs encountered in real-world applications.

All these files are stored alongside the benchmark runner, when VNS needs to execute a program it will call the benchmark runner which will execute the program and return the minimum execution time.

To analyse our results we will be tabling our findings and drawing graphs using matplotlib to observe performance and convergence.

In conclusion of our background, using VNS , a fast-converging effective metaheuristic coupled with a local search derives solutions from two well established mechanisms that have been proven to do well in optimisation problems.

Implementation

Overview

In this implementation section, we will look deeper at the mechanisms employed for Parallel Variable Neighbourhood search integrated with hill climbing. Our implementation goal is to create a robust framework in Python to fulfil all our requirements.

Our implementation leverages two parallel VNS methods and one standalone VNS method to navigate the combinatorial challenge posed by compiler flag optimisation.

Note, that all pseudocode listed are high-level abstractions of the actual implemented code, they are used to convey the logic and understanding of the functions implemented, without delving into the intricacies.

Flag encoding

To encode our flags as bitstrings, a JSON file has been created in the following format:

```
{
...
"-fno-gcse-after-reload": false,
"-fno-ipa-cp-clone": false,
"-fno-loop-interchange": false,
"-fno-loop-unroll-and-jam": false
...
}
```

Displayed are flags included within the O3 preset, following the design choice, they are set to false, indicating that they are inactive, in bitstring terms would be represented as False,False,False,False or 0000. Running the GCC/G++ compiler with this setup would produce

```
gcc -O3 benchmark_program.c -o output_file
```

After running our algorithm we will observe changes with the state

```
{
...
"-fno-gcse-after-reload": true,
"-fno-ipa-cp-clone": false,
"-fno-loop-interchange": true,
"-fno-loop-unroll-and-jam": false
...
}
```

Now the flags correspond to the 1010 bitstring and when executing with this configuration the GCC/G++ compiler would use the command

```
gcc -O3 benchmark_program.c -o output_file -fno-gcse-after-reload -fno-loop-interchange
```

As denoted by the -fno prefix, these flags are deactivated. The effect of O3 in this command would be all O3 flags excluding those explicitly disabled, this mechanism provides precise control over which optimisations are applied.

A bit flip is done by the following code

```
flags[flag_key] = not flags[flag_key]
```

Variable Neighbourhood Search

Our standalone VNS method (VNS_{SEQ}) has been designed to have three main components: a shake, local search and neighbourhood change.

```
FUCNTION variable_neighbourhood_search(args):
fastest_state <- initial_state
fastest_exec_time <- execute(fastest_state)
k = 1

while {until condition satifised} do
shake(initial_state, k)
new_execution_time <- execute(shaken_state)
if new_execution_time < fastest_exec_time:
        fastest_exec_time <- new_execution_time
        k = 1
...
```

We begin with the initialisation, where we assign the initial_state to the fastest_state, the initial_state will always be all flags disabled in the JSON file, this represents -O3 being enabled, we execute this to get an initial execution time. This sets a benchmark for improvement.

We define k to be 1, k in this case is the neighbourhood_size.

The loop continues until a termination condition is met. This could be a limit on the number of iterations, a time constraint, or a satisfactory performance threshold, in our case we set it until 30 iterations, since we did not notice a notable improvement of using N//2 over it, where N is the number of flags, which is 174.

The shake function is designed to perturb the current solution by making ‘k’ random changes to the flags The intensity of these perturbations is controlled by the parameter k, which is dynamically adjusted throughout the program. Initially, k is set to 1, leading to minimal disruption to the solution to explore the immediate neighbourhood.

The shake function returns a shaken state. To evaluate the efficacy of this new state we execute it to obtain its execution time. This is then compared to the fastest execution time. If an improvement is found, the we update our variables to reflect this new optimal state, and the k value is reset to 1 to intensify the search around this newly found promising area.

```
FUCNTION shake(args):
keys <- shaken_state.keys()
last_shaken_keys <- []
randomise(keys)
for key in keys[:k] do
        shaken_state[key] <- not shaken_state[key]
        < method for handling flags with options >
        last_shaken_keys.append(key)
return shaken_state, last_shaken_keys
```

In our shake function, we begin by identifying all keys that are not already enabled in the state, thus qualifying them for potential modification. After randomising the keys we perform a bit flip on the first k number keys. This slicing operation, denoted by [:k], limits our focus to the initial k elements in the randomised list of keys. Each key that undergoes this bit flip. We then append shaken keys to the list variable last_shaken_keys. This step ensures that there is no immediate reversion of changes made during the shake in the subsequent local search phase, allowing that the exploration be meaningful.

Now that our state is shaken, its execution time measured and assessed whether it presents an improvement over our current best. We can continue our VNS algorithm, by now implementing local search, the second part of our algorithm.

```
FUCNTION variable_neighbourhood_search(args):
...
state, exec_time, improvement_found <- hill_climbing_local_search(args)
```

Local search

```
FUCNTION hill_climbing_local_search(args):
randomise(keys)
iteration_counter <- 0
for key in keys do
    if iteration_counts == 10:
        break
    if key in last_shaken_keys:
        continue

    neighbour_state <- shaken_state.copy()
    neighbour_state[key] <- not shaken_state[key]

    < method for handling flags with options >

    execution_time <- exec(neighbour_state)


    if execution_time < current_best_time:

        return neighbour_state

    iteration_counter += 1
```

Our simple hill climbing algorithm firstly sets an iteration_counter to 0, this is as we designed our algorithm to only visit 10 neighbours before terminating, this was so we could cut down on execution times whilst sacrificing minimal performance.

We randomise the keys and iterate through them, we exclude any keys that have just been shaken to avoid reverting changes introduced during the previous shaking phase, this ensures that the shake's impact is given a chance to be evaluated properly before any potential reversal.

In our neighbour_state we toggle the flag corresponding to the current key, this is a bit flip, the toggle represents a neighbour of our shaken_state, which is then explored by the exec() function. If the execution time is better than our current_best_time then we've found a more optimal set of flags and break out of the search with an improvement found.

However if we do not find a more optimal set of flags then our iteration_counter is incremented until it reaches a predefined amount, in this case 10, and then our local search will terminate without improvement.

```
FUCNTION variable_neighbourhood_search(args):
...
if improvement found:
        fastest_exec_time <- local_search_exec_time
        fastest_state <- local_search_state
        k <- 1
else:
        k <- k + 1
...
```

After our local search terminates we are returned whether or not an improvement was found, if so, then we update our variables accordingly and reset the neighbourhood search size, k, to one, this concentrates the search on our newly found best state. Otherwise we increment our neighbourhood search size.

Going back to local search, we have also implemented a steepest ascent hill climbing, it was tested to be inferior to simple hill climbing in our context and therefore not used in our main algorithm. However the logic of it follows the same as simple hill climbing except a few changes to its exit conditions

After our for loop iterates through all the keys, an exit condition is stated: if an improvement is found, then exit, this condition is placed outside the loop, ensuring we return the best solution at the end.

Benchmarking Mechanism

Our execution command (exec()) is signed as run_gcc_with_benchmark, the responsibility of this method is to take all flags and build a base command, which is the GCC/G++ compiler command and run command, which is the compiled file run. This is all timed using the usr/bin/time program.

The arguments are important to mentioned for this method, it takes two: benchmark_file and flags_to_omit.

benchmark_file specifies the .c file used to execute the set of flags against, as mentioned we have a collection of 14.

The flags_to_add is a list of flags that are to be added following the GCC command , this list is created by using list comprehension to take all flags in the current_state that have ‘True’ assigned as the value. The first 111 flags have -fno prefixed since they are a part of the -O3 preset, hence enabling them will deactivate said flag, the last 63 optimisation flags are prefixed with -f, enabling them will activate them.

```
FUCNTION run_gcc_with_benchmark(args):

unique_exec_name <- f"benchmark_program_{uuid}"
config <- get_program_config(args)

base_command <- f"gcc -O3 {benchmark_file} -o {unique_exec_name} {compile_flags}
for flag in flags_to_add:
        based_command += f" -f{flag}"

subprocess.run(based_command)

execution_times = []
for i -> 3 do
```

```
            subprocess.run(run_command)
            execution_times.append(stdout.strip())

return min(execution_times)
```

Our benchmark function firstly utilises the uuid package to create unique names for the output executable, this is especially important when running in parallel, as using the same name for the output executable will lead to clashes between cores attempting to delete and run it simultaneously.

The run command and compiler flags are benchmark program specific, and are retrieved from the get_program_config method which in turn retrieves all data from a JSON file to return, the JSON file contain program specific commands is listed like so:

```
  "floyd-warhsall": {
    "compile_flags" :"-I src/polybench-c-3.2/utilities",
    "run_command": "/usr/bin/time -f '%e' ./{exec_name}",
    "input_value": None ,
  },
  "binary-trees": {
     "compile_flags": "-lm",
     "run_command": "/usr/bin/time -f '%e' ./{exec_name} 19",
     "input_value": None
```

The compiler flags appended to the end of the GCC/G++ command are used to link the program with external libraries. The run command also varies depending on the inputs.

Once all information for the benchmark program has been retrieved we are able construct the base command, flags are then added on the based command.

We use the subprocess module to run our commands on the operating system, once the base command has been compiled a new executable will be present in our working directory. A new array to hold all execution times is initiated and we run the base command three times.

To reduce effects of noise we run it three times and take the minimum execution time produce, this is attained with the in-built min() command, which is returned.

Parallel VNS cont.

Multiple Homogenous VNS Processes – VNS_{HP}

Inspired by ParamILS, we initiate independent VNS processes. The random nature of shaking mean there is a 1/174 chance of two processes having the same state after a shake, regardless the random neighbour selection for local search and subsequent searches make it improbable for two processes to follow the same path. Having two cores run separate instances of our algorithm allow us to explore more of the search space, potentially returning a better solution that VNS_{SEQ}.

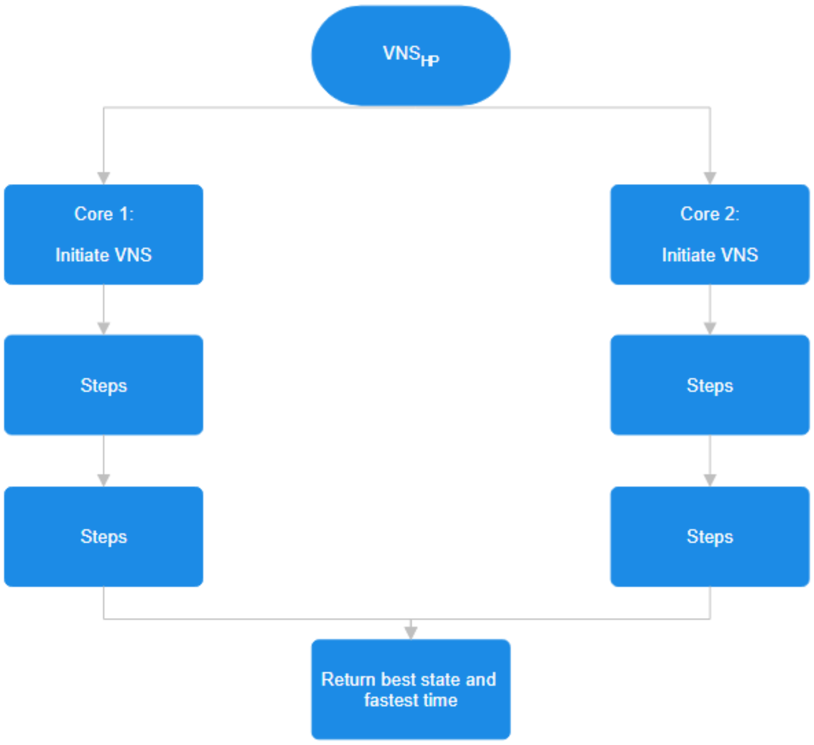


Figure 8: Flow state chart depicting core functionality of parallel VNShp

To implement this parallel functionality we will utilise the concurrent.futures module from Python's standard library. This module provides a high-level interface for asynchronously executing callables using pools of threads or processes.

```
FUNCTION vns_wrapper(args):

fastest_state, fastest_exec_time = initalise()

return fastest_state, fastest_exec_time
```

We first define a wrapper function that wraps the VNS algorithm for ease of use in parralel execution, it takes the initial state of the search, the benchmark file path and number of cores to be used. It calls an initialise function that handles the initial setups and then runs the variable neighbourhood search.

```
with concurrent.futures.ProcessPoolExecutor(max_workers=num_processes) as executor:

futures <- [executor.submit(args, num_processes) for _ in range(num_processes)]

results <- [f.result() for f in concurrent.futures.as_completed(futures)]
```

Our parallel VNS method will set up a process pool executor that manages a pool of worker processes to execute calls asynchronously with the ProcessPoolExectuor method. Following this we submit tasks to the executor, the VNS wrapper is called multiple times here equal to the number of cores specified. Having a VNS wrapper encapsulates the initialisation and execution of the VNS algorithm as well as providing the ability to work in concurrency due to it being a self-contained unit of “work”.

Once we are done with execution we collect all the results from the futures as they complete, the as_completed method returns an iterator objects that yields futures as they complete, but calling .result on them we can get the return values from vns_wrapper. This is then returned and we choose the best state based on the execution time.

Perhaps, the more complicated parallelisation technique added is the farmer-worker model

To have more success in discovering around a state, each process performs a local search independently from a state, they explore different paths from the same starting point. After the local search ends, the findings are consolidated and the best execution is chosen.

Due to the nature of parallelization, we had to abandon the simple hill climbing strategy and adopt a steepest ascent hill climbing, where the best possible solution from those executed is chosen. The farmer-worker model operates where the farmers initialises the process, sets up the initial state and distributes different parts of the search space to each worker for a broad exploration. The results are consolidated and the algorithm proceeds accordingly.

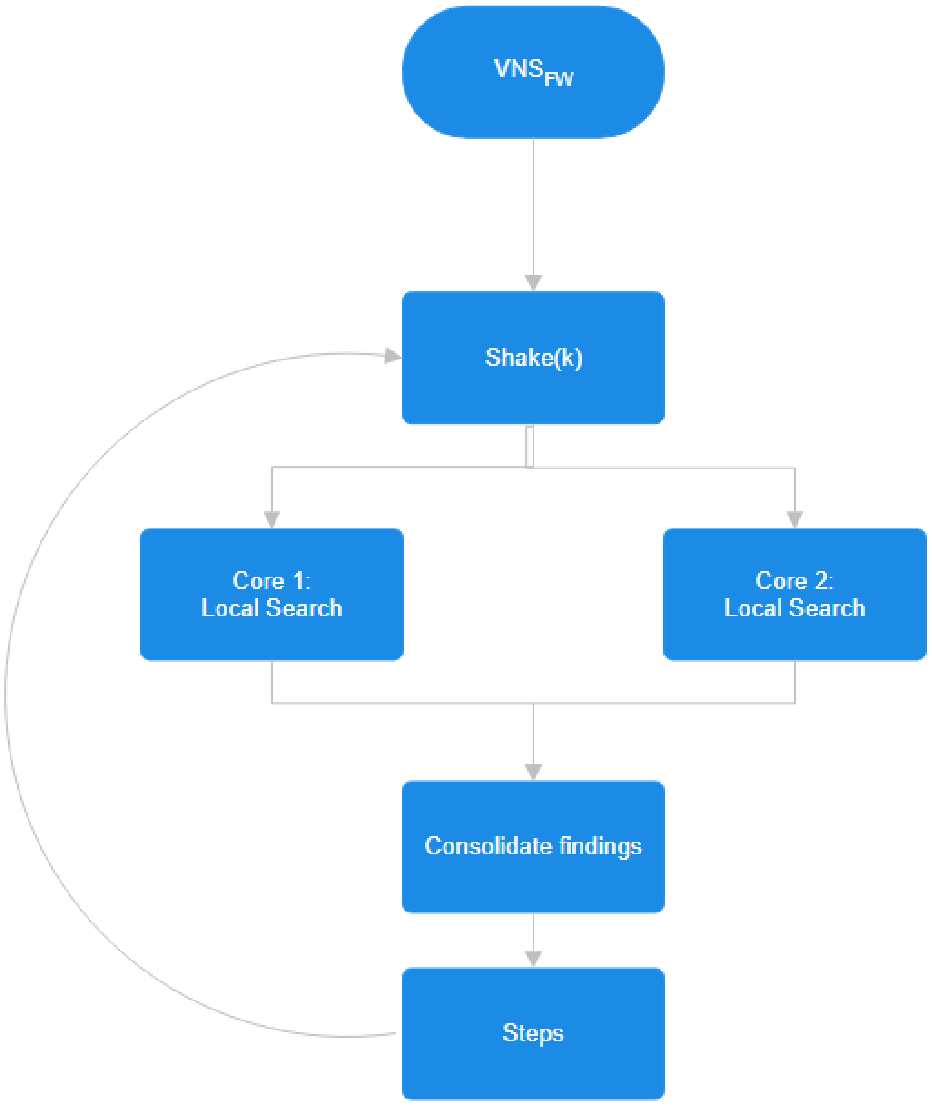


Figure 9: Flow state chart depicting core functionality of parallel VNSfw

Each worker calls the explore_neighbour method, this method essentially performs a bit flip, and produces a list of flags_to_add, it then executes the perturbed state and returns the execution time along with the flag settings to the farmer, which then compares it against the best known execution time.

The hill_climbing_local_search_parrallel function calls the worker, this function acts as the father, coordinating the parallel exploration of the search space. It , like the VNS$_{HP}$ uses the concurrent.futures library for parallel execution.

```
total_runs <- num_workers * (12 / num_workers)
```

Firstly, we have the initialisation, besides the setup of common variables like best_time , best_neighbour etc. this method introduces the total_run variable , which calculates the total number of runs by distributing 12 tasks equally among the workers. Each worker will handle 3 tasks (12 divided by the number of workers).

```
task_keys <- [key for key in keys if key not in last_shaken_keys]

random.shuffle(task_keys)  # random shuffle for diverse exploration
```

Like our local searches we filter out keys from consideration if they were just shaken , and then shuffle them to introduce randomisation into the selection process

```
with concurrent.futures.ProcessPoolExecutor(max_workers=num_workers) as executor:

    tasks = [(args) for key in task_keys[:total_runs]]

    futures = [executor.submit(explore_neighbor, task) for task in tasks]
```

Just like the VNS_{HP} we use the ProcessPoolExecutor to manage a pool of worker processes, then generate a list of tasks where each task is a tuple containing necessary arguments for exploring a neighbour, the tasks are limited to the number calculated in total_runs, which is always 12. The tasks are then submitted to the executor for parralel execution, where each task is handled by the explore_neighbour function

```
for future in concurrent.futures.as_completed(futures):
```

As the futures complete we iterate through them, retrieving each result of the futures, which include the key explored, execution time achieved and the modified flags states. The farmer will check if the new execution time is better than the known best. If so, it updates the best times, marks the corresponding neighbour, sets the improvement found to True, breaks out of the loop and return shaken_state, best_time, improvement_found. The rest of the process is the same as VNS_{SEQ}.

Additional utilities have been implemented into our framework to aid with evaluating our project, and with flags having an option choice, although this is not relevant to our main algorithm, so will not be explained in detail.

Experimentation and Evaluation

Variable Neighbourhood search analysis

We will now begin testing to ensure that the theoretical implementation of our algorithms translate into real-world performance gains. This section outlines our approach to experimentally validating our variable neighbourhood search.

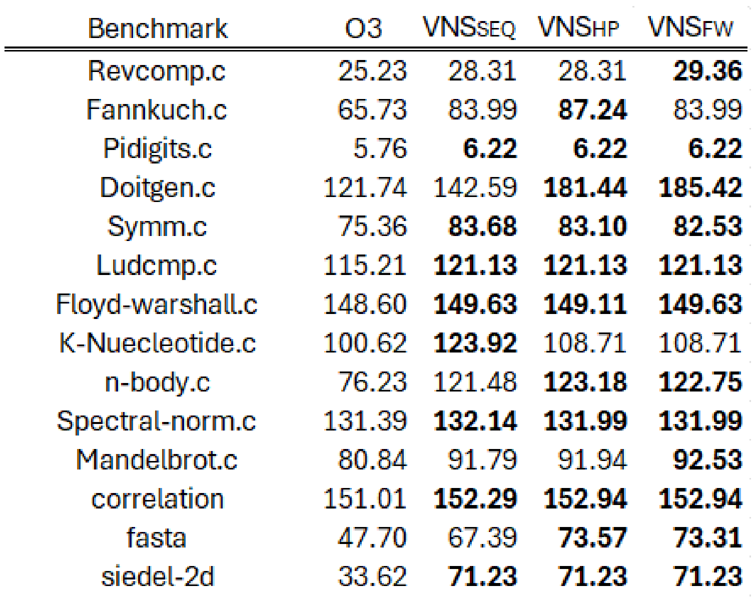
To test our parallel, all programs will be run on an Amazon AWS t2.medium, which contains 2 vCPUs and 8GB of RAM, there are no background programs running, allowing the process to run uninterrupted until the stopping condition is met. For VNS_{SEQ} however we will shift to a t1.small instance, which contains 1 vCPU and 4GB of RAM, this adjustment ensures that the t2.medium's capabilities do not disproportionately enhance VNS_{SEQ}'s performance.

We compare our algorithm against 3 static and commonly applied compiler configurations:

- None: No flags are activated
- O3: -O3 flags are activated

We present our results against the none configuration calculating the percentage difference between them, a larger value indicates a greater performance increase. To calculate the percentage difference we use the following formula

$$P\% = \frac{|v_1 - v_2|}{\left(\frac{(v_1 + v_2)}{2}\right)} \times 100$$



Benchmark	O3	VNS$_{SEQ}$	VNS$_{HP}$	VNS$_{FW}$
Revcomp.c	25.23	28.31	28.31	**29.36**
Fannkuch.c	65.73	83.99	**87.24**	83.99
Pidigits.c	5.76	**6.22**	**6.22**	**6.22**
Doitgen.c	121.74	142.59	**181.44**	**185.42**
Symm.c	75.36	**83.68**	**83.10**	**82.53**
Ludcmp.c	115.21	**121.13**	**121.13**	**121.13**
Floyd-warshall.c	148.60	**149.63**	**149.11**	**149.63**
K-Nuecleotide.c	100.62	**123.92**	108.71	108.71
n-body.c	76.23	121.48	**123.18**	**122.75**
Spectral-norm.c	131.39	**132.14**	**131.99**	**131.99**
Mandelbrot.c	80.84	91.79	91.94	**92.53**
correlation	151.01	**152.29**	**152.94**	**152.94**
fasta	47.70	67.39	**73.57**	**73.31**
siedel-2d	33.62	**71.23**	**71.23**	**71.23**

Figure 10: Performance chart showing VNS algorithms against NONE and O3 configurations

Firstly, as expected the O3 flag provides a large performance boost against the None configuration in all programs.

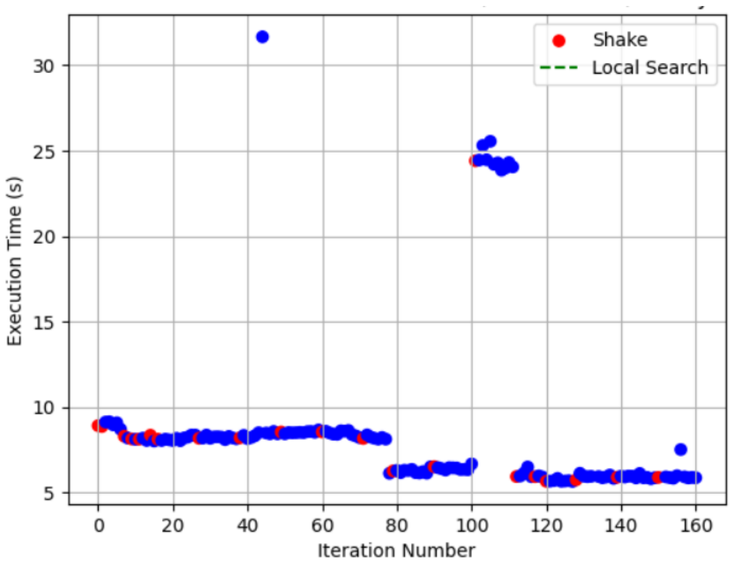
Our basic VNS algorithm outperforms O3 in all instances, averaging an 18% performance increase. For specific programs such as seidel-2d, fasta, fannkuch, doitgen, n-body there is a substantial performance improvement relative to -O3 (25%+), as already investigated we know that O3 does not activate all the flags that are mentioned in the documentation as the prior ALL configuration was outperformed by it in all programs. The difference against O3 varies with data-mining algorithm, correlation.c only seeing a 0.8% performance increase, which was only 0.3 seconds, whereas stencil program, seidel-2d showed a 70%+ performance. The substantial reduction in execution times achieved by our VNS algorithm compared to the O3 preset confirms our initial hypothesis: O3 is not the optimal flag preset for all programs, but rather a general one.

With respect to the NONE configuration, our algorithm also outperform it by more than 50%+ in 12/14 benchmark programs used.

In some other cases, although there is a time reduction, it is minimal, this is apparent with floyd-warshall.and spectral-norm.

Looking at our various VNS variants, in general ,parallelised VNS seems to boast a marginal performance boost over our traditional non- parallelised VNS counterpart, the average performance increase is 3.8% for VNS$_{HP}$ and 4.0% for VNS$_{FW}$, the 0.2% difference between the two algorithms is negligible due to slight noise in the system, despite attempts to mitigate it. Overall a ~4.0% increase with our parallelised variants is still notable and beneficial as these gains can be quite important for time critical systems. One outlier to point out would be the kernel algorithm, doitgen.c where a ~25% improvement was noted. This was due to the sequential non-parallelised VNS algorithm being unable to explore the complete search space effectively, both parallel variants managing to find the improvement inform us that they both have similar capabilities and the choice of choosing one would be down to the program you are trying to optimise.

Examining the line graph generated by our algorithm for two specific programs, we can visualise the important aspects of the variable neighbourhood search (VNS). The red dots represent the shaking events and blue dots – indicating local searches - cluster around these shakes. The 1-Hamming distance concentrated local search contrasted with our shakes that will be changing dozens of bits per shake create this effect. You can see some pretty significant jumps at times, these occur when the 'k' value of VNS is adjusted to a larger number, signalling a major changed in the neighbourhood size. The local search acts as a dynamic and effective searching mechanism, consistently driving improvements. This is evident from the strong correlation observed in the graph between the intensity of shakes and subsequent local search improvements.



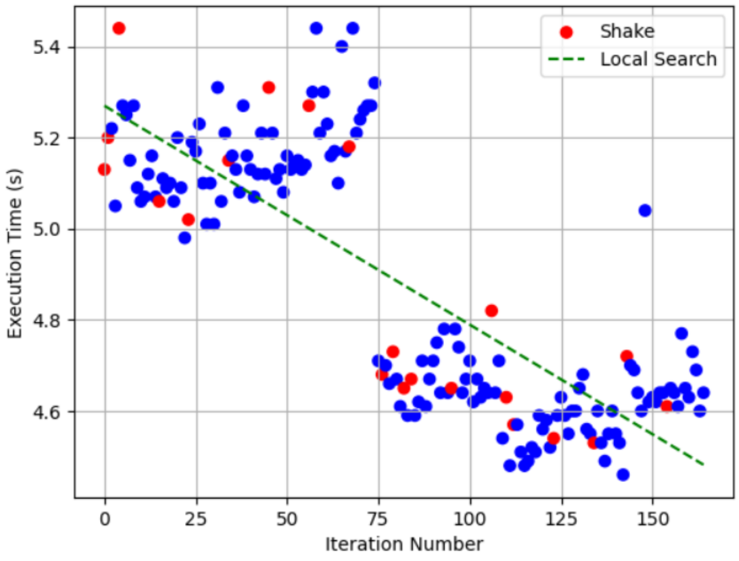
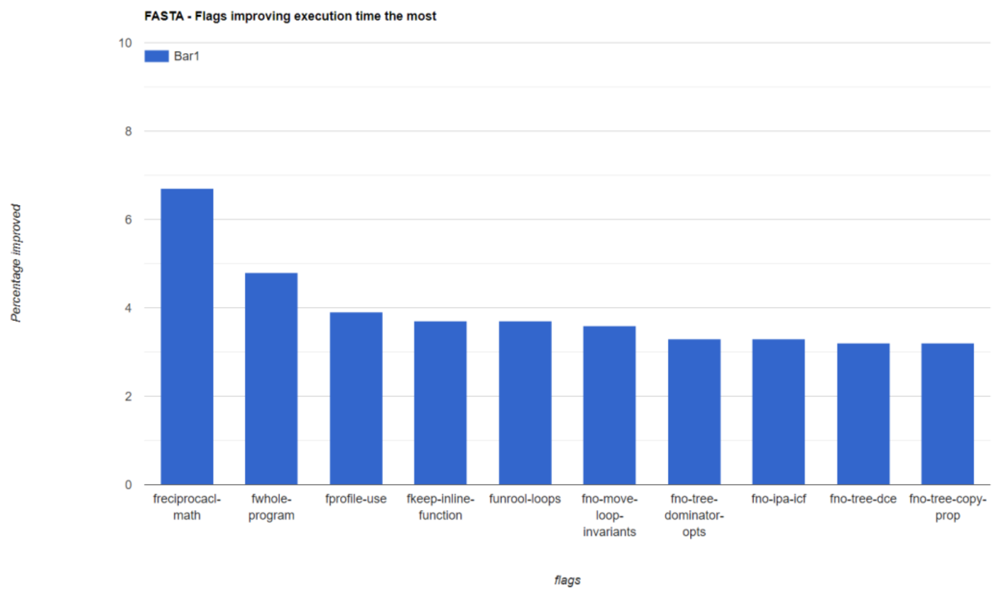
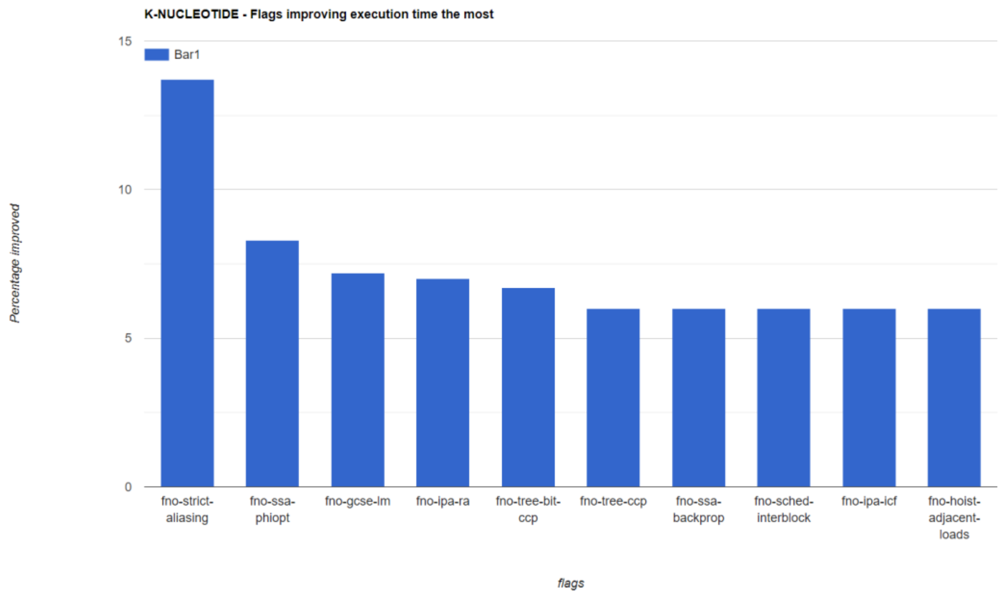


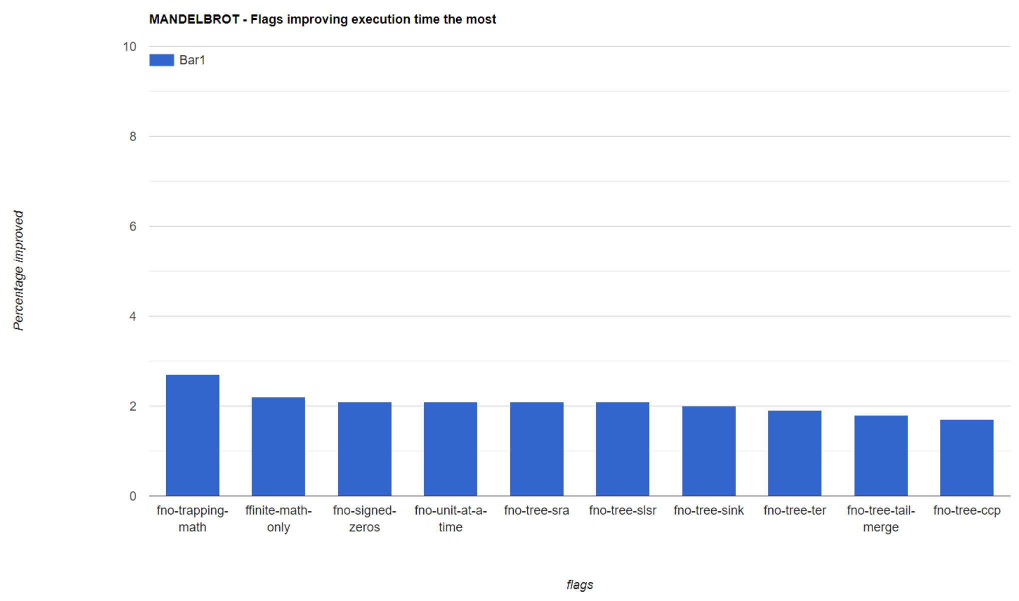
Figure 11: VNS progression of n-body.c and fannkuch.c displayed

Individual Flag Analysis

Looking deeper at the flags used and their individual impact on each program, we iteratively ran each flag with the GCC/G++ compiler measuring it's impact. We did this for four programs.







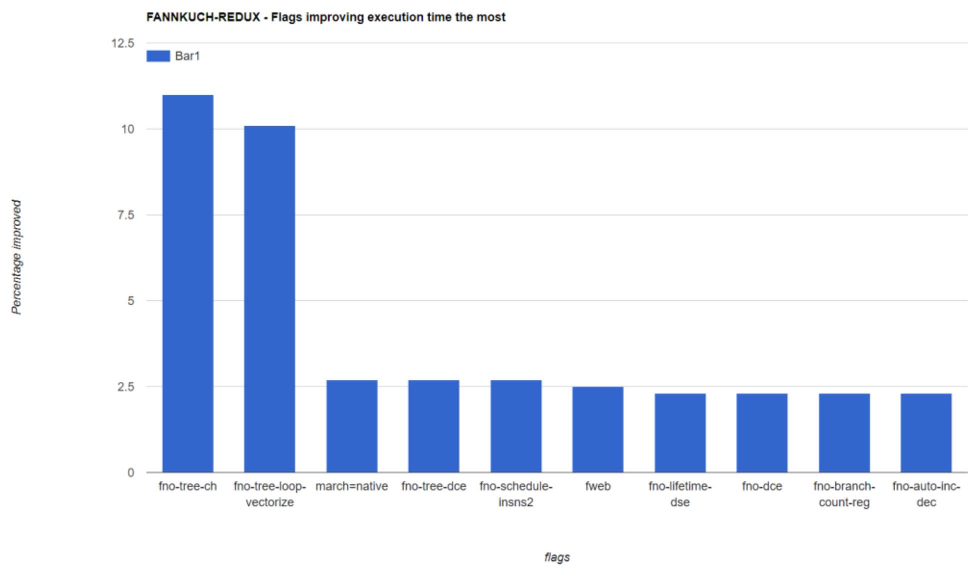


Figure 12: Shows flags and their performance improvement based no four programs

Above are four graphs depicting the flags that had the greatest performance increase of our programs (i.e decreasing execution time). From our analysis, the majority of compiler flags that showed improvement were those prefixed with -fno-, indicating that removing flags from the -O3 preset often yielded more benefit than adding additional flags. This observation suggests that the default -O3 configuration may be over-optimised for certain applications, incorporating flags that have a detrimental effect on our program performance. This observation makes sense, given that the -O3 preset is designed to be a general optimisation setting that is suitable for a broad range of programs, the inevitable inclusion of optimisations that, while generally beneficial, may degrade performance in certain cases. This variability provides the foundation for our project, allowing us to explore more tailored optimisation strategies that potentially outperform the one-size-fits-all approach of the -O3 preset.

Lastly, we ran the similar algorithm for additional programs and then averaged out the results.

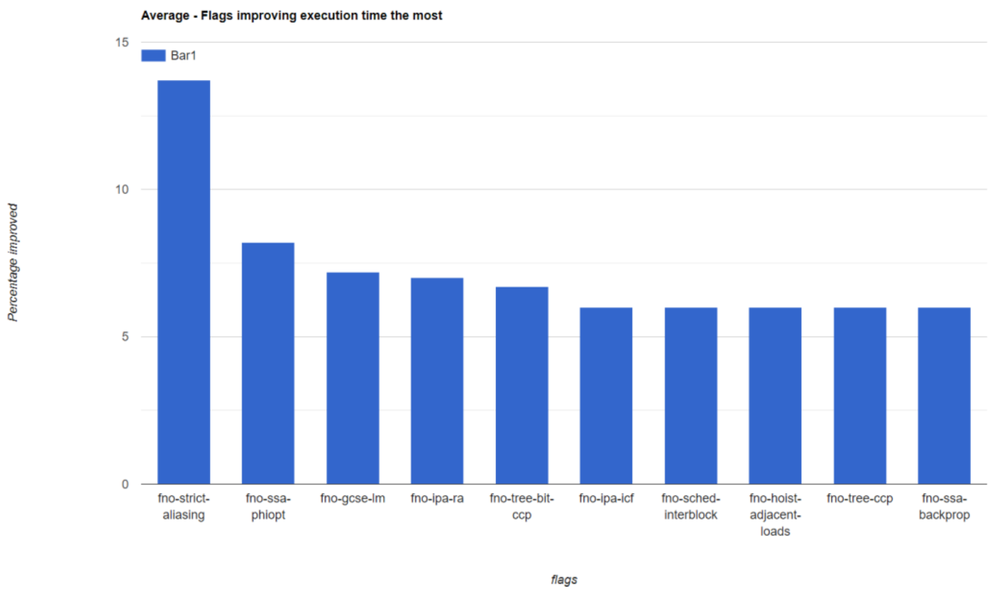


Figure 13: Shows which actions have the best effect in general for all programs

Out of the list of all flags and their individual impact on the program, we identified 101 -fno flags that improved program performance, some flags had minimal impact on the program with a <2% performance increase, this could be due to noise or potentially overhead costs by the individual flags.

The most beneficial action on average was removing the -fno-strict-aliasing, which is the flag that enables the compiler to make assumptions about how different object types access memory, based on strict aliasing rules. This flag increases instruction parallelism, reduces memory traffic, enhances vectorisation and optimises memory access. The farrago of optimisations provided by a single flag may cause a lot of overhead due to the amount of potential optimisations it attempts.

Generally, this analysis also proved that the approach of mainly removing flags whilst having -O3 enabled before compilation is a very viable solution to our problem.

Professional and ethical issues

Environmental and societal factors

The average data centre produces 177,000,000 kW-hr of electricity, uses 227,000,000L of water and requires hundreds of thousands of kgs of metal and other synthetic materials to be built. [34] Data centres account for 2.5% to 3.7% of global GHG emissions, which exceeds that from the aviation industry. [35]

It is clear that despite the advantages of cloud computing, it’s most notable drawback is the seemingly unlimited processing power available at the click of a few buttons. This convenience for users can lead to inefficient resource usage since they do not own the hardware and due to overestimating hardware requirements. In context of running C code, incorrect flag settings can increases CPU usage. By applying our algorithm, which optimises these setting, we can on average reduce the execution time by approximately 20% compared to standard O3 compilation. If 10% of computers run C code in 10 datacentres, a 20% less execution time could save more than 30,000,000 kW-hr of electricity. This is just for 10 datacentres, there are approximately nearly 11,000 data centre locations worldwide, the potential for reducing carbon emissions based on simple flag optimisation would have profound societal and environmental benefits.

Code of Conduct.

Throughout the development of this project, we adhered to the guidelines and standards established by the British Computing Society. This project is structured in a way that it does not support or enable unethical activities. Furthermore, it is developed with careful consideration for privacy, security, well-being of others, and most importantly environmental impact. [36]

Conclusion

The results show without ambiguity that our optimisation algorithm outperforms the O3 flag preset for every single benchmark program. Coming into this project we predicated it on the hypothesis that O3 is not a one-size-fits-all optimisation, and that selectively removing flags would benefit our execution time.

Our evaluations confirm this assumption ,while some flags introduced detrimental overhead or adversely affect performance in other ways, further research is needed to precisely identify these effects.

An average performance increase of 18% from O3 is a substantial improvement that would benefit even for applications where time is not a critical factor. Our solution also incorporates robust error checking to prevent the application or removal of flags that could disrupt program functionality, providing the reliability that developers expect from the O2 preset.

If I were to undertake this project again or had additional time, I would change the strategies employed in this project. Despite the success of VNS, it was primarily chosen due to (a) its resemblance to genetic algorithms, which have already been documented widely for their efficacy in compiler optimisation tasks, and (b) the choice was both ambitious and more innovative. With the successful implementation of VNS, I would be more ambitious to experimenting with more advanced variants of VNS, such as General VNS integrated with techniques like Simulated Annealing to enhance the adaptive search capabilities. Additionally, experimenting with different local search strategies that involve hyperparameter tuning could provide deeper insights into the optimisation landscape.

Another fascinating direction would be the integration of machine learning to predict the efficacy of compiler flags based on hotspots. However, gathering a substantial dataset would have not been feasible to gather within the timeframe. Creating such as dataset would be an excellent goal for a longitudinal study.

Overall, the projects success opens numerous avenues for additional research and development especially in the field of VNS.

References

[1] "Options That Control Optimization." GCC, the GNU Compiler Collection. Available at: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[2][5] et. al., E. P. . (2021). Experimental Analysis Of Optimization Flags In Gcc. Turkish Journal of Computer and Mathematics Education (TURCOMAT), 12(7), 1875–1879. Retrieved from https://turcomat.org/index.php/turkbilmat/article/view/3088

[3] D. Branco and P. R. Henriques, "Impact of GCC optimization levels in energy consumption during C/C++ program execution," 2015 IEEE 13th International Scientific Conference on Informatics, Poprad, Slovakia, 2015, pp. 52-56, doi: 10.1109/Informatics.2015.7377807.

[4] Börstler, J., Bennin, K.E., Hooshangi, S. et al. Developers talking about code quality. Empir Software Eng 28, 128 (2023). https://doi.org/10.1007/s10664-023-10381-0

[6] Nicholas FitzRoy-Dale. "Benefits of Compiler Optimisation." NICTA and The University of New South Wales, Sydney, Australia. Available at: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=8ac77ff99e6ed5db06327ef87617865b3b805519

[7] B. Tağtekin, B. Höke, M. K. Sezer and M. U. Öztürk, "FOGA: Flag Optimization with Genetic Algorithm," 2021 International Conference on INnovations in Intelligent SysTems and Applications (INISTA), Kocaeli, Turkey, 2021, pp. 1-6, doi: 10.1109/INISTA52262.2021.9548573.

[8] T. Sandran, M. N. B. Zakaria and A. J. Pal, "A Genetic Algorithm approach towards compiler flag selection based on compilation and execution duration," 2012 International Conference on Computer & Information Science (ICCIS), Kuala Lumpur, Malaysia, 2012, pp. 270-274, doi: 10.1109/ICCISci.2012.6297252.

[9] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. "Optimizing for Reduced Code Space using Genetic Algorithms." Available at: https://dl.acm.org/doi/pdf/10.1145/314403.314414.

[10] "Informed Search Algorithms." Available at: https://web.pdx.edu/~arhodes/ai9.pdf.

[11] Stuart Russell and Peter Norvig, "Artificial Intelligence: A Modern Approach," Third Edition. Available at: https://people.engr.tamu.edu/guni/csce421/files/AI_Russell_Norvig.pdf.

[12] Bart Selman and Carla P. Gomes, "Hill-climbing Search," Cornell University, Ithaca, New York, USA. Available at: https://www.cs.cornell.edu/selman/papers/pdf/02.encycl-hillclimbing.pdf.

[13] Siby Abraham, Imre Kiss, Sugata Sanyal, and Mukund Sanglikar, "Steepest Ascent Hill Climbing For A Mathematical Problem," arXiv preprint arXiv:1010.0298. Available at: https://arxiv.org/abs/1010.0298

[14] Brototi Mondal, Kousik Dasgupta, Paramartha Dutta, Load Balancing in Cloud Computing using Stochastic Hill Climbing-A Soft Computing Approach, Procedia Technology, Volume 4, 2012, Pages 783-789, ISSN 2212-0173, https://doi.org/10.1016/j.protcy.2012.05.128. (https://www.sciencedirect.com/science/article/pii/S2212017312004070)

[15] E. . -G. Talbi and T. Muntean, "Hill-climbing, simulated annealing and genetic algorithms: a comparative study and application to the mapping problem," [1993] Proceedings of the Twenty-sixth Hawaii International Conference on System Sciences, Wailea, HI, USA, 1993, pp. 565-573 vol.2, doi: 10.1109/HICSS.1993.284069

[16] Gabriel Luque and Enrique Alba, "Finding Best Compiler Options for Critical Software Using Parallel Algorithms," available at: https://riuma.uma.es/xmlui/bitstream/handle/10630/16910/idc.pdf?sequence=3&isAllowed=y

[17] Hansen, Pierre & Mladenovic, Nenad & Moreno-Pérez, José. (2010). Variable neighbourhood search: Methods and applications. 4OR. 175. 367-407. 10.1007/s10479-009-0657-6.

[18] Hansen, P., Mladenović, N. (2018). Variable Neighborhood Search. In: Martí, R., Pardalos, P., Resende, M. (eds) Handbook of Heuristics. Springer, Cham. https://doi.org/10.1007/978-3-319-07124-4_19

[19] Mladenovic, Nenad. (2004). A Tutorial on Variable Neighborhood Search. https://www.researchgate.net/publication/2906122_A_Tutorial_on_Variable_Neighborhood_Search

[20] Rahim Mammadli, Ali Jannesari, and Felix Wolf, "Static Neural Compiler Optimization via Deep Reinforcement Learning, available at: https://arxiv.org/pdf/2008.08951.pdf

[21] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?," 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, Italy, 2018, pp. 612-621, doi: 10.1109/SANER.2018.8330266

[22] [25] Prathibha A. Ballal, H. Sarojadevi, and Harsha P S, "Compiler Optimization: A Genetic Algorithm Approach," International Journal of Computer Applications, Volume 112, Number 10, available at: https://research.ijcaonline.org/volume112/number10/pxc3900938.pdf.

[23] T. Sandran, M. N. B. Zakaria and A. J. Pal, "A Genetic Algorithm approach towards compiler flag selection based on compilation and execution duration," 2012 International Conference on Computer & Information Science (ICCIS), Kuala Lumpur, Malaysia, 2012, pp. 270-274, doi: 10.1109/ICCISci.2012.6297252

[24] B. Tağtekin, B. Höke, M. K. Sezer and M. U. Öztürk, "FOGA: Flag Optimization with Genetic Algorithm," 2021 International Conference on INnovations in Intelligent SysTems and Applications (INISTA), Kocaeli, Turkey, 2021, pp. 1-6, doi: 10.1109/INISTA52262.2021.9548573.

[26] Sandran, T., Zakaria, N., Pal, A.J. (2012). An Optimized Tuning of Genetic Algorithm Parameters in Compiler Flag Selection Based on Compilation and Execution Duration. In: Deep, K., Nagar, A., Pant, M., Bansal, J. (eds) Proceedings of the International Conference on Soft Computing for Problem Solving (SocProS 2011) December 20-22, 2011. Advances in Intelligent and Soft Computing, vol 131. Springer, New Delhi. https://doi.org/10.1007/978-81-322-0491-6_55

[27] Keith D. Cooper, Timothy J. Harvey, and Jeff Sandoval, "Tuning an Adaptive Compiler," January 2007, available at: https://www.researchgate.net/publication/228757857_Tuning_an_Adaptive_Compiler

[28] G. R. Reddy, K. Madkour and Y. Makris, "Machine Learning-Based Hotspot Detection: Fallacies, Pitfalls and Marching Orders," 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Westminster, CO, USA, 2019, pp. 1-8, doi: 10.1109/ICCAD45719.2019.8942128.

[29] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?," 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, Italy, 2018, pp. 612-621, doi: 10.1109/SANER.2018.8330266

[30] Hansen, P., Mladenović, N. & Moreno Pérez, J.A. Variable neighbourhood search: methods and applications. Ann Oper Res 175, 367–407 (2010). https://doi.org/10.1007/s10479-009-0657-6

[31] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown, "ParamILS: An Automatic Algorithm Configuration Framework," Journal of Artificial Intelligence Research, available at: https://www.jair.org/index.php/jair/article/view/10628/25415

[32] The Computer Language Benchmarks Game (2023) https://sschakraborty.github.io/benchmark/measurements-gpp.html

[33] Polybench programs https://web.cs.ucla.edu/~pouchet/software/polybench

[34] Dennis Bouley, "Estimating a Data Center's Electrical Carbon Footprint," APC by Schneider Electric, available at: https://sg.insight.com/content/dam/insight/en_US/pdfs/apc/apc-estimating-data-centers-carbon-footprint.pdf.

[35] Georgette Kilgore, "Carbon Footprint of Data Centers," 8 Billion Trees, available at: https://8billiontrees.com/carbon-offsets-credits/carbon-ecological-footprint-calculators/carbon-footprint-of-data-centers

[36] https://www.bcs.org/media/2211/bcs-code-of-conduct.pdf