

Design level software optimization of structural analysis tool through memoization and application of concurrent computing methods

Myron Tsatsarakis

Thesis submitted in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Associate Prof. *Polyvios Pratikakis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

**Design level software optimization of structural analysis tool through
memoization and application of concurrent computing methods**

Thesis submitted by
Myron Tsatsarakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: *Myron Tsatsarakis*
Myron Tsatsarakis

Committee approvals: _____
Polyvios Pratikakis
Associate Professor, Thesis Supervisor

Angelos Bilas
Professor, Committee Member

Kostas Magoutis
Associate Professor, Committee Member

Departmental approval: _____
Polyvios Pratikakis
Associate Professor, Director of Graduate Studies

Heraklion, July 2022

Design level software optimization of structural analysis tool through memoization and application of concurrent computing methods

Abstract

Performance is an important concern for the end user of an application. Latency, unresponsiveness of commands and long wait times can make or break the experience. Some of the factors that prevent software engineers from achieving their performance goals are the presence of legacy code and the inability to exploit all the available hardware resources.

Legacy code acts as a barrier that prevents both feature extensibility and optimization-oriented refactoring. It is considered to be a no-man's-land, preventing any fruitful discussion about its detailed functionality and side effects in performance. On the other hand, hardware resources, such as CPU cores, can be exploited to reduce the time cost of computationally expensive operations by the application of concurrency. Data-oriented design for concurrency can provide an increase in performance by distributing workload among cores, fully utilizing the available hardware.

Our work is applied on a commercial structural design and analysis application which suffers from the aforementioned setbacks. It comes with a large code-base, guaranteeing a sizeable number of legacy modules. Furthermore, efficient hardware utilization, was not possible during the early development times of the application since the technological research regarding concurrent computation was not sufficiently developed.

In the first part of this thesis we present a way to memoize the return values of computationally expensive legacy code operations by designing an extensible Memoization Model. We identify the computationally expensive operations by conducting performance analysis using an external profiling tool and apply our Memoization Model to the legacy code base in a non-intrusive way. We present the requirements, design and implementations of our Memoization Model using C++. Our evaluation indicates that the application of our Memoization Model, yields up to 100% increase in performance and has been scheduled for a commercial release, in a future update of the application.

In the second part of this thesis we present a way to concurrently calculate computationally expensive properties of structural elements. We extend the design of the manager component, responsible for calculating these expensive properties, by applying concurrency semantics. We use the concept of asynchronous tasks and our own implementation of a thread pool to aid our design. We present the requirements, design and implementation of our concurrent manager component using C++. Our evaluation indicates that our concurrent manager component, yields up to 420% increase in performance and has been scheduled for a commercial release, in a future update of the application.

Σχεδιαστική βελτιστοποίηση εφαρμογής στατικών κτηριακών μελετών μέσω μεθόδων υπομνηματισμού και κατανεμημένου υπολογισμού

Περίληψη

Η απόδοση μια εφαρμογής είναι σημαντικός παράγοντας για τον τελικό χρήστη. Καθυστερήσεις, εντολές χωρίς ανταπόκριση και μεγάλες περίοδοι αναμονής υποβαθμίζουν την εμπειρία χρήσης μιας εφαρμογής. Μερικοί από τους λόγους που εμποδίζουν τους μηχανικούς λογισμικού από το να βελτιώσουν την απόδοση των εφαρμογών είναι η ύπαρξη παλαιωμένου κώδικα και η ανεπαρκής εκμετάλλευση πόρων υλισμικού.

Ο παλαιωμένος κώδικας αποτελεί εμπόδιο στην επεκτασιμότητα των δυνατοτήτων μιας εφαρμογής και στην αναδόμηση του κώδικά της με σκοπό τη βελτιστοποίηση. Η έλλειψη τεκμηρίωσής του εμποδίζει συζητήσεις σχετικά με τη λειτουργία και τις παρενέργειες του στο σύστημα. Από την άλλη, πόροι υλισμικού όπως οι επεξεργαστικοί πυρήνες μπορούν να εκμεταλλευτούν κατάλληλα ώστε να μειωθεί ο χρόνος των υπολογιστικά ακριβών λειτουργιών, μέσω μεθόδων κατανεμημένου υπολογισμού. Εφαρμόζουμε τη δουλειά μας σε μία εμπορική εφαρμογή στατικών κτηριακών μελετών η οποία πάσχει από αυτά τα προβλήματα. Περιέχει μία μεγάλη βάση κώδικα με αρκετές παλαιωμένες δομικές ενότητες. Επιπλέον, η επαρκής εκμετάλλευση των πόρων υλισμικού δεν ήταν δυνατή κατά τη συγγραφή της εφαρμογής, καθώς η τεχνολογία σε εργαλεία κατανεμημένου υπολογισμού δεν ήταν αρκετά αναπτυγμένη.

Στο πρώτο μέρος της εργασίας αυτής παρουσιάζουμε ένα τρόπο υπομνηματισμού τιμών υπολογιστικά ακριβών και παλαιωμένων λειτουργιών, σχεδιάζοντας ένα επεκτάσιμο μοντέλο υπομνηματισμού. Ανακαλύπτουμε τις υπολογιστικά ακριβείς λειτουργίες αναλύοντας την απόδοση της εφαρμογής και εφαρμόζουμε το μοντέλο υπομνηματισμού σε παλαιωμένο κώδικα με μη παρεμβατικό τρόπο. Παρουσιάζουμε τις απαιτήσεις, τη σχεδίαση και την υλοποίηση του μοντέλου υπομνηματισμού σε γλώσσα C++ . Η αξιολόγηση της απόδοσης της δουλειάς μας δείχνει πως η εφαρμογή του μοντέλου υπομνηματισμού προσφέρει μέχρι και 100% αύξηση της απόδοσης. Η εφαρμογή του μοντέλου έχει προγραμματιστεί για εμπορική χρήση μέσω της ένταξής του σε μελλοντική αναβάθμισή της εφαρμογής.

Στο δεύτερο μέρος της εργασίας αυτής παρουσιάζουμε ένα τρόπο κατανεμημένου υπολογισμού υπολογιστικά ακριβών ιδιοτήτων κατασκευαστικών στοιχείων. Επεκτείνουμε τη σχεδίαση του στοιχείου διαχείρισης υπολογισμού των ιδιοτήτων αυτών εφαρμόζοντας μεθόδους κατανεμημένου υπολογισμού. Χρησιμοποιούμε τις έννοιες ασύγχρονων διεργασιών και υλοποιούμε μια ομάδα νημάτων ως βοηθητικά εργαλεία της σχεδίασής μας. Παρουσιάζουμε τις απαιτήσεις, τη σχεδίαση και την υλοποίηση του κατανεμημένου στοιχείου διαχείρισης σε γλώσσα C++ . Η αξιολόγηση της απόδοσης της δουλειάς μας δείχνει πως το κατανεμημένο στοιχείο διαχείρισης προσφέρει μέχρι και 420% αύξηση της απόδοσης. Το κατανεμημένο στοιχείο διαχείρισης έχει προγραμματιστεί για εμπορική χρήση μέσω της ένταξής του σε μελλοντική αναβάθμισή της εφαρμογής.

Acknowledgements

First of all, I would like to thank my advisor, Associate Prof. Pratikakis Polyvios for his guidance and support. His advice helped me in planning ahead during uncertain times regarding this work. I also thank him for providing me with the opportunity of conducting a work that is not only research relevant, but also industrially applicable.

I would also like to thank my advisors and senior staff members of TOL, Mr. Tsagarakis Manos and Mr. Babukas Emmanuil for pushing this work to commercial availability as well as for their technical support and advice.

Special thanks to my dear friends and colleagues Giortamis Manos and Lydakias Giorgos. Our companionship and life perspective helped me grow as a person.

Furthermore, I am grateful to my family for their encouragement, patience, trust and mental support. Lastly I would like to thank all my close and international friends for our shared memories together. These memories helped me keep my spirits up during difficult times.

στους γονείς μου

Contents

1	Introduction	1
2	Solver Memoization	7
2.1	Background	7
2.2	Methods	8
2.2.1	Performance Profiling	8
2.2.2	Memory layout of the Solver database	10
2.2.3	The iteration pattern CPU Bottleneck	10
2.2.4	Complexity Analysis of the iteration pattern	14
2.2.5	The Memoization Model Requirements	15
2.2.6	The Memoization Model Interface	16
2.2.7	The Memoization Model Hash Table Implementation	17
2.2.8	The Memoization Model Trie Implementation	19
2.2.9	Linking our work with the code base	23
2.3	Evaluation	23
2.3.1	The input file collection	25
2.3.2	Contents of an input file	25
2.3.3	The input file generator script	26
2.3.4	Regression Testing	26
2.3.5	Performance Evaluation	28
2.3.6	Evaluation on NUMA architecture Virtual Machine	28
2.3.7	Evaluation on the user PC machine	31
2.4	Related Work	35
2.5	Future Work	35
3	Capacity Volume Manager Concurrency	37
3.1	Background	37
3.2	Methods	39
3.2.1	The Linear Design	39
3.2.2	Concurrent Design Requirements	41
3.2.3	The Concurrent Design Implementation	43
3.2.4	The Thread Pool Implementation	47
3.3	Evaluation	50

3.3.1	The Testing Pipeline	50
3.3.2	Regression Testing	54
3.3.3	Performance Evaluation	54
3.3.4	Evaluation on NUMA architecture Virtual Machine	55
3.3.5	Evaluation on the user PC machine	59
3.4	Related Work	64
3.5	Future Work	65
4	Conclusion	67
	Bibliography	69

List of Figures

1.1	The RAF user interface	3
2.1	Visual Studio CPU usage report	9
2.2	Beam Distributed Load Element layout in memory	11
2.3	Memory format of Beam Distributed Load Elements in FORTRAN	11
2.4	The <code>GetNumber</code> function	12
2.5	The <code>GetAttributes</code> function	13
2.6	Usage pattern of <code>GetNumber</code> and <code>GetAttributes</code> functions in the code base	14
2.7	FORTRAN to C interoperability function signature	16
2.8	The value signature of a mapping of <i>CachedLoadsMap</i>	17
2.9	The interface signatures	17
2.10	The <code>UnorderedMapImpl</code> class	18
2.11	The <code>TrieImpl</code> class signature	20
2.12	The <code>TrieImpl</code> class methods	21
2.13	Optimally constructed Trie in terms of node size	22
2.14	The new <code>GetNumber</code> and <code>GetAttributes</code> functions.	24
2.15	A 10 x 10 x 10 generated structural model	27
2.16	Comparing two versions of the same output file	27
2.17	Memoization Model <i>speedup</i> on all user input files	29
2.18	Memoization Model <i>speedup</i> on all generated input files	30
2.19	Memoization Model execution time on user input	32
2.20	Memoization Model execution time on generated input	33
2.21	Memoization Model <i>speedup</i> on all inputs	34
3.1	Graphical representation of Capacity Volume	38
3.2	Capacity Volume Calculation Progress bar	38
3.3	Capacity Volume Key format	39
3.4	The <code>OnionManager</code> class	40
3.5	The Concurrent <code>OnionManager</code> class signature	44
3.6	The <code>RequestOnion</code> method	45
3.7	The <code>GetOnionConcurrent</code> method	46
3.8	The <code>GetOnionSolve</code> method	47
3.9	The <code>ThreadPool</code> class signature	48

3.10	The <code>ThreadPool</code> class methods	49
3.11	The <code>GenerateKeys</code> function	51
3.12	The <code>RunBenchmark</code> function	52
3.13	The <code>ScopeTimer</code> class	53
3.14	<i>Speedup</i> of Concurrent Capacity Volume Manager implementations	56
3.15	<i>Speedup</i> of <code>CONCURRENT_POOL</code> implementation	57
3.16	<i>Efficiency</i> of <code>THREAD_POOL</code> implementation	58
3.17	<i>Speedup</i> of <code>THREAD_POOL</code> implementation on 1612 requests with error bars	59
3.18	<i>Speedup</i> of Concurrent Capacity Volume Manager implementations	60
3.19	Execution time of Concurrent Capacity Volume Manager implemen- tations	61
3.20	Comparison of time lost due synchronization while a single thread is running	62
3.21	<i>Speedup</i> of <code>THREAD_POOL</code> implementation	63
3.22	<i>Efficiency</i> of <code>THREAD_POOL</code> implementation	64

Chapter 1

Introduction

Performance is an important concern for the end user of an application. Latency, unresponsiveness of commands and long wait times can make or break the experience. Civil engineers are a user base which favors performance in their go-to software. They expect structural design and analysis software to be responsive to user inputs and have minimal wait times during CPU intensive operations. Reality can be often disappointing, since commercial structural analysis and design tool suites are too complicated to develop while being void of performance problems. Industry level software like AutoCAD from Autodesk [3] or STAAD by Research Engineers International (REL) [29] can exhibit case specific performance issues that lead to the frustration of thousands of users. However, some performance problems can be attributed to specific factors.

One of those factors is legacy code bloat. It is a phenomenon present in any kind of commercial application, from independently developed small scale plugins to industry level tool suites. Legacy code is characterized by lack of unit tests, no documentation, dependency hell, spaghetti code flow, comments that misinform intent and by many more frustrating issues. It can impede the performance of an application by providing a hard limit to code optimization. Software engineers can improve the performance of a module by optimizing it's code. However, if that module is dependent on legacy code, then the application of optimization techniques becomes limited, since a legacy code base acts as a no man's land. Furthermore, a legacy code module, that acts as a dependency to many other modules, extends the coverage of the problem to a larger part of the application, further limiting the actions of engineers.

A different factor that can impede performance is CPU utilization. Most industry level applications developed over fifteen years ago have been designed with serial execution in mind. However, with the modern rise of parallel programming, concurrent architectures and methods of writing mutually exclusive code have been developed. Developments in research regarding concurrent computation has essentially altered people's perception of software engineering as a practice. It made people realize that old code can be more efficiently rewritten to better utilize hardware resources, creating the need of upgrading multiple old architectures to new ones that favor concurrency.

Structural design and analysis applications face both of those issues. Their long development history guarantees the existence of legacy code that impacts performance in a substantial way. Furthermore, the mutually exclusive nature of computations applied in structural elements leads to the need of more efficiently utilizing hardware resources, to balance out the computational cost among processors.

It is difficult to deal with the issues above in a foolproof manner. People working with the legacy part of a code base tend to modify the code as little as possible in order to minimize the possibility of unwanted bugs. The need of introducing parallel computation without having the time and resources to delve into the theoretical parts of concurrency fundamentals leads to diminishing results. Usage of multiprocessing and concurrency API's to parallelize small parts of a code base, without conducting proper performance analysis, rarely leads to meaningful results.

Both of these approaches do not strike at the heart of the problem. Legacy code is not modernized. It is still regarded as legacy code with applied performance fixes. Using tools that favor concurrency without basic knowledge of the fundamentals does not lead to efficient CPU utilization neither to substantial performance increases.

A more meaningful approach would be to refactor the problematic architectures and present modern extensible ones that favor performance. Use of abstractions and the application of generic programming leads to the construction of extensible interfaces and intuitive tools. Other engineers can use these resources as blueprints to refactor more and more parts of a code base with similar problematic architectures.

In this work, we present the way we profile, analyze and optimize the architecture of two legacy code-rich modules of a structural design and analysis application for civil engineers. The results of this work are going to be commercially available, since they have been scheduled for a future update of the product.

We apply our work on RAF [30], a structural design and analysis application developed by Technical Software House (TOL) [10], a company which specializes in the development of static building analysis software. The development of RAF goes back to more than 21 years, guaranteeing the existence of a large legacy code base. After conducting talks with the architects of RAF, we reached the conclusion that there is a demand for an increase in performance on two modules

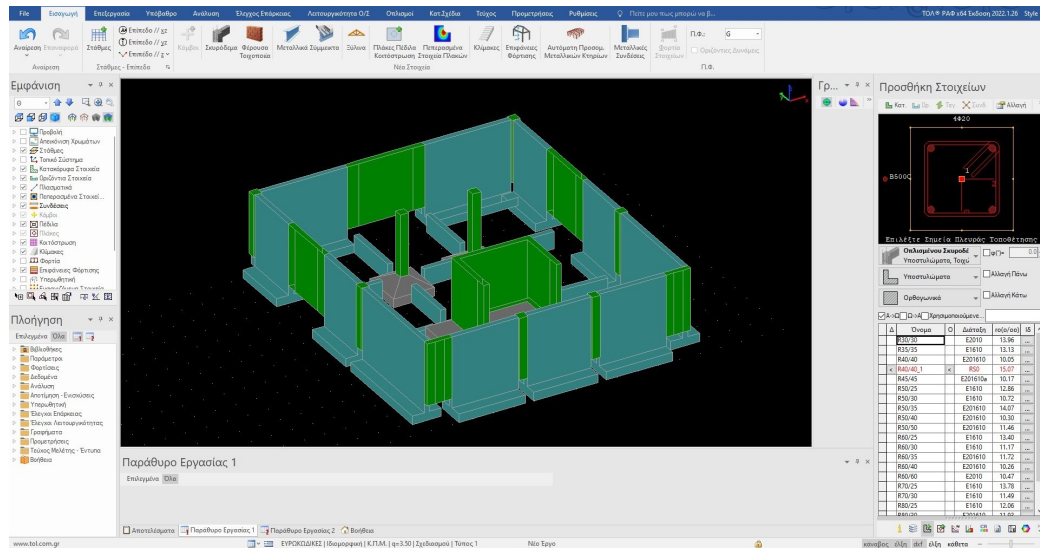


Figure 1.1: The RAF user interface

of the application. These modules suffer from the aforementioned problems, them being legacy code bloat that prevents extensibility, and potential for application of concurrency in the architectural level.

These modules are the Solver and the Capacity Volume Manager. The Solver is a command line tool which performs structural analysis, in the form of CPU intensive calculations, on a structural model to ensure its safety and structural integrity. The Capacity Volume Manager is responsible for calculating and storing the Capacity Volumes of structural elements. The Capacity Volume of an element is a computationally expensive property that represents it's resistance against high pressure, strong winds, earthquakes and other phenomena.

The architects of RAF have noticed that the Solver module has been taking an unusually large amount of time to Solve structural models of medium to large size. After applying embarrassing-parallelism optimizations using the OpenMP tool, regarding data transferring, they did not notice a substantial increase in performance. It is clear that a performance analysis of the module is required.

Our work regarding the Solver module consists of

- Profiling the performance of the solver using various structural models as input.
- Analyzing the legacy code parts that contribute to a performance bottleneck.
- Designing, implementing and injecting a generic solution for the performance problem.
- Testing and evaluating the results which lead up to a 2.0 *speedup* ratio in performance.

First we gather a collection of medium to large size input files of structural models. These input files represent user generated, as well as tool generated, structural models. The first ones allow us to conduct our research based on real life structural engineering models and produce results observable by the typical end user. The second ones help us focus the validation of our work on specific test cases.

Using the Visual Studio Profiler, we benchmark the CPU usage of various executions of the Solver. After analyzing the output of the profiler, we notice that up to 75% of CPU time is dedicated to a specific iteration pattern, with linear time asymptotic complexity, present in various functions of the code base. The legacy code of the module along with the fact that these functions are present in multiple call trees during execution, makes it impossible to modify the application control flow leading to these calls.

Instead, we propose a new design that eliminates the linear complexity of this pattern. After injecting an initialization step, we replace the old iteration pattern with a new Memoization Model, resulting in a constant time complexity. We notice a theoretical and practically confirmed reduction in the execution time of the Solver module when we compare the constant time complexity of the new model with the linear complexity of the old pattern. The new model relies on a key-value store, for which we propose two different implementations, one using a Hash Table and another using a Trie data structure.

After successfully tackling the performance issue, we are presented with the problems of regression testing the functionality of the solver and of profiling the resulted *speedup* in execution time. For regression testing, we compare the contents of output files we generate using the old iteration pattern and the new Memoization Model. Finally, we benchmark the execution time by comparing the total execution time of the Solver module before and after applying our work, using the same structural model as input. We notice up to a 2.0 *speedup* ratio in execution time among our input files. The new Memoization Model has already been merged in the main development branch of the Solver module and has been scheduled for a future release.

The Capacity Volume Manager provides methods which allow the user to request an instance of a Capacity Volume of a structural element. If the Capacity Volume is already calculated, it is immediately returned. Otherwise, the Capacity Volume must be calculated before being returned. The calculation of a single Capacity Volume takes a considerable amount of time, while multiple instances of Capacity Volume calculations share zero state among them. This raises the possibility of batch calculating multiple Capacity Volumes concurrently.

Our work regarding the Capacity Volume Manager module consists of

- Understanding and analyzing the design of the Capacity Volume Manager.
- Designing, implementing and injecting an architecture which allows for concurrent requests of Capacity Volumes.

- Testing and evaluating the results which lead up to a 4.2 *speedup* ratio in performance.

Initially, we analyze the design of the Capacity Volume Manager and focus on the operation that allows the requests of Capacity Volumes. We call the method that matches this operation **GetOnion**, since onion is slang for Capacity Volume among structural engineers, due to the onion-like graphical representation of a Capacity Volume. **GetOnion** allows the user to communicate with the Capacity Volume Manager interface and request Capacity Volumes. Since we cannot modify the established interface to a large extent, we treat the signature of **GetOnion** as an invariant while designing our solution.

GetOnion returns the requested Capacity Volume to the user. If the requested Capacity Volume is not already calculated upon request, then it is calculated on demand before being returned. Calculating the Capacity Volume is a computationally expensive operation which can be concurrently executed for multiple Capacity Volumes. We extend the functionality of the Capacity Volume Manager by redesigning the control flow of **GetOnion**. We use mutex objects to enable concurrent calculation of Capacity Volumes, while preventing data races and enabling mutual exclusion of critical sections.

After introducing concurrency semantics to the Capacity Volume Manager, we are presented with the question of how to handle thread management. We present two different solutions, one of them using the asynchronous facilities of the STL, the other one using a thread pool of our own implementation. Both these solutions along with the new design of the Capacity Volume Manager share the same STL futures/promises semantics.

Lastly, we implement our own testing and benchmarking suite and use it to generate a large amount of Capacity Volume requests concurrently. We use this suite to conduct concurrency, regression and coverage tests of our new concurrent design. Finally, we benchmark the old and new design of the Capacity Volume Manager, noticing up to a 4.2 *speedup* ratio on the same collection of requests. The Concurrent Capacity Volume Manager has already been merged in the main development branch and has been scheduled for a future release.

The rest of the text is organised as follows

- Our work is split in two parts, the first one regarding our work related to the Solver module, the second one regarding our work related to the Capacity Volume Manager module
- On each part, we offer a background section, where we give information about the context our work is applied on.
- Then, we describe the methods and processes used to conduct our work.
- Afterwards, we present the way we evaluate our work in terms of testing and performance, as well as present our results.

- Finally, we present other research related to our work as well as possible ways to extend it.

Chapter 2

Solver Memoization

2.1 Background

The Solver is one of the many modules of the RAF tool suite. It is a command line tool, integrated with the UI of the main RAF application. The Solver module helps structural engineers in estimating the seismic performance of existing reinforced concrete buildings.

In particular, the Solver evaluates the inelastic seismic response of reinforced concrete buildings by analyzing the characteristics of reinforced concrete sections while also accounting for the effects of distributed plasticity. The Solver mainly performs linear algebra computations to calculate the freedom matrices of structural components. The freedom matrix describes a component's resistance to external forces. To calculate the freedom matrix, the Cholesky Decomposition algorithm is used [5]. The Solver tool is partial implementation of *A computational tool for evaluation of seismic performance of reinforced concrete buildings* by *S.K.Kunnath, A.M.Reinhorn, J.F.Abel* [15].

As input, the Solver module accepts two text files. The first one describes the schema of a structural model and has a .XSD suffix. The second one includes the contents of the structural model itself and has a .XML suffix. The Solver creates C++ data bindings using the .XSD schema and parses the .XML input file using CodeSynthesis XSD, a Schema to C++ data binding compiler [6]. As output, the Solver produces a text file containing the analysis results which has a .ROU suffix.

The execution of the Solver module is comprised of 4 distinct steps

1. Parse the structural model from the .XML input file.
2. Convert the model to a compatible memory format.
3. Solve the model, in terms of performing seismic analysis.
4. Produce the output text file.

The Solver tool has been written in the FORTRAN programming language and compiled using the Intel FORTRAN Compiler [12]. During the time of our work,

the architects of RAF are in the process of modernizing the module by rewriting its code using the C++ language. In order to future proof our work, we have used C++ and the FORTRAN to C interoperability routines [13] to inject our code to the pre-existing FORTRAN legacy code base.

To perform seismic analysis, by solving an input model on step 3, the Solver must determine the motion vector \vec{U} of the building model. This is determined by multiplying the inverse of the freedom matrix K to the external forces vector \vec{F} . To calculate the freedom matrix, the Cholesky Decomposition algorithm is used, which is considered a time consuming operation.

$$\vec{U} = K^{-1}\vec{F}$$

The architects of RAF have noticed that the Solver takes significant time to solve small to medium size structural models. They suspected that the the memory conversion on step 2 acts as a performance bottleneck. Most of the work on step 2 consists of iterations of copy-transforming a vector to another format, which is considered an embarrassingly parallel problem. They used the OpenMP API [23] to distribute the work among threads. However, the results did not live up to a significant *speedup* of the Solver’s execution time. It is clear that a performance analysis of the Solver is needed to identify potential performance bottlenecks.

2.2 Methods

2.2.1 Performance Profiling

To profile the Solver tool we collected user provided input files. These files contain a structural model produced by users of RAF. The architects of RAF consider these inputs to take an unusually large amount of time to be solved. We profile a complete execution of the Solver tool, from parsing the input file to producing the output file.

To do this, we use the Visual Studio 2019 CPU Usage Diagnostic Tool [19]. From now on we will refer to this tool as Visual Studio Profiler. The Visual Studio Profiler collects information about the functions that are executing in an application and provides a list of them, ordered in descending CPU usage time. After we pick the function with the highest percentage of *Self CPU* usage and dive into its call tree, we notice that most of the CPU time is not spent on the Cholesky decomposition method as one would expect, but on a specific iteration pattern. This pattern is prevalent on utility functions that provide the user with information about the Beam Distributed Load Elements and Beam Concentrated Force Load Elements of the structural model.

As we observe the Visual Studio Profiler results in figure 2.1, 20% of CPU time is spent at the iteration loop shown at the code section of the screen. This loop

Current View: Functions <input type="button" value="v"/>			
Function Name	Total CPU [unit, ...]	Self CPU [unit, %]	Module
__schr_common_main_seh	387357 (99.78%)	0 (0.00%)	RAFMainCPP.exe
fem::raf::dispatcher::solve	387347 (99.78%)	0 (0.00%)	RAFMainCPP.exe
TOLLIBRAF_MAIN	383092 (98.68%)	0 (0.00%)	TOLLibRAF.dll
SOLVE	382768 (98.60%)	0 (0.00%)	TOLLibRAF.dll
POST_STATIC_PROCEDURES	257192 (66.25%)	0 (0.00%)	TOLLibRAF.dll
MEMBER_SEGMENT_INTERNAL_FORCES_MAN	257180 (66.25%)	112 (0.03%)	TOLLibRAF.dll
X_POSITION_INTERNAL_LCL_FRCS_DSPS	257054 (66.22%)	609 (0.16%)	TOLLibRAF.dll
MDL_BEAM_AUXIL_RESULTS_mp_GET_PARAMETE...	165988 (42.76%)	165818 (42.71%)	TOLLibRAF.dll
MDL_BEAM_AUXIL_RESULTS_mp_GET_NUMBER_E...	120098 (30.94%)	119980 (30.91%)	TOLLibRAF.dll
STATIC_SOLUTION	71999 (18.55%)	0 (0.00%)	TOLLibRAF.dll
MDL_TOTAL_STRUCTURE_BALANCE_mp_NODES_...	53577 (13.80%)	88 (0.02%)	TOLLibRAF.dll
SOLVE_BUNDLE	52087 (13.42%)	0 (0.00%)	TOLLibRAF.dll
MDL_INTERNAL_SOLVE_mp_INTERNAL_SOLVE	52087 (13.42%)	0 (0.00%)	TOLLibRAF.dll
CHOLESKY_DECOMPOSITION	51780 (13.34%)	51738 (13.33%)	TOLLibRAF.dll
DISTRIBUTED_LOAD_BEAM_ELEMENT_ONE_ALL	38153 (9.83%)	97 (0.02%)	TOLLibRAF.dll
DISTR_ELEM_LOAD_IDENTIFIER_SNEW	37126 (9.56%)	37102 (9.56%)	TOLLibRAF.dll
MDL_BEAM_LOAD_mp_GET_LOCAL_LOAD_BEAM...	36107 (9.30%)	5 (0.00%)	TOLLibRAF.dll
MDL_BEAM_LOAD_mp_GET_GLOBAL_LOAD_BEAM...	35478 (9.14%)	14 (0.00%)	TOLLibRAF.dll
MDL_NODES_BALANCE_mp_GET_TOTAL_SYSTEM...	17963 (4.63%)	3 (0.00%)	TOLLibRAF.dll

C:\MyronTsaRepos\RAFSolver.myrontsa\F90_RAF\TOLLibRAF\source\mdl_beam_auxil_results.f90:358			
81084 (20.89%)	402	do i=1, di%dl%DISTR_LOADS_NUMBER	
1 (0.00%)	403	if (di%dl%DISTR_LOADS_ELEM(i) == ielem) then	
114 (0.03%)	404	if (di%dl%DISTR_LOADS_LC(i) == icase) then	
93 (0.02%)	405	if (di%dl%DISTR_LOADS_KIND(i) == iload) then	
30 (0.01%)	406	count = count + 1;	
83 (0.02%)	407	if (count == idl) then	
11 (0.00%)	408	index = i;	
	409	exit;	
	410	end if	
	411	end if	
	412	end if	
	413	end if	
84296 (21.71%)	414	end do	

Figure 2.1: Visual Studio CPU usage report

pattern is present in at least two other functions named `MDL_BEAM_AUXIL_RESULTS_mp_GET_PARAMETERS` and `MDL_BEAM_AUXIL_RESULTS_mp_GET_NUMBER_ELEMENTS`, as shown at the profiler results window.

The columns shown at the profiler results window are the function name, *Total CPU* and *Self CPU* respectively. *Total CPU* indicates how much CPU time was spent by the function and any functions called by it. High *Total CPU* values point to the functions that are most expensive overall. *Self CPU* indicates how much time was spent in the function body, excluding the time spent on functions that were called by it. High *Self CPU* values may indicate a performance bottleneck within the function itself.

The iteration pattern is considered a CPU bottleneck since it corresponds to a large value of *Self CPU* time.

2.2.2 Memory layout of the Solver database

Before we dive into the details of the iteration pattern, we need to understand how Beam Distributed Load Elements and Beam Concentrated Force Load Elements are represented in memory.

After studying the corresponding FORTRAN header files we understand that Beam Distributed Load Elements and Beam Concentrated Force Load Elements share the same attributes and memory layout. Given this, we only need to analyze the layout of Beam Distributed Load Elements. Each element consists of 5 attributes that describe it in its entirety. The names of these attributes are relevant to their structural properties. Since we consider the structural engineering element irrelevant to our work, we abstract their names to `attr0`, `attr1`, `attr2`, `attr3`, `attr4`. `attr0`, `attr1`, `attr2` are one dimensional dynamically allocated integer sized arrays. `attr3`, `attr4` are two dimensional dynamically allocated floating point sized arrays, with the second dimension having a constant size of 2. We can think of `attr3`, `attr4` as having the same layout with the rest of the attributes, with the difference being that each element has the size of two floating points instead of one integer. The Beam Distributed Load Elements format and layout in the Solver memory database are shown in figure 2.2 and figure 2.3.

2.2.3 The iteration pattern CPU Bottleneck

The problematic iteration format is present in various parts of the code base. Two functions that contain this pattern have had a significant impact in CPU time. They are used to retrieve a utility value from the `BeamDistributedLoads` Solver database and are called many times in the application code.

The iteration loop parses the `BeamDistributedLoads` Solver database the number of Beam Distributed Load Elements that match the function parameters. One of the functions that utilizes this iteration returns the above count. Another function returns attributes of an element that matches the above count. We will abstract the names of these functions to `GetNumber` and `GetAttributes`.

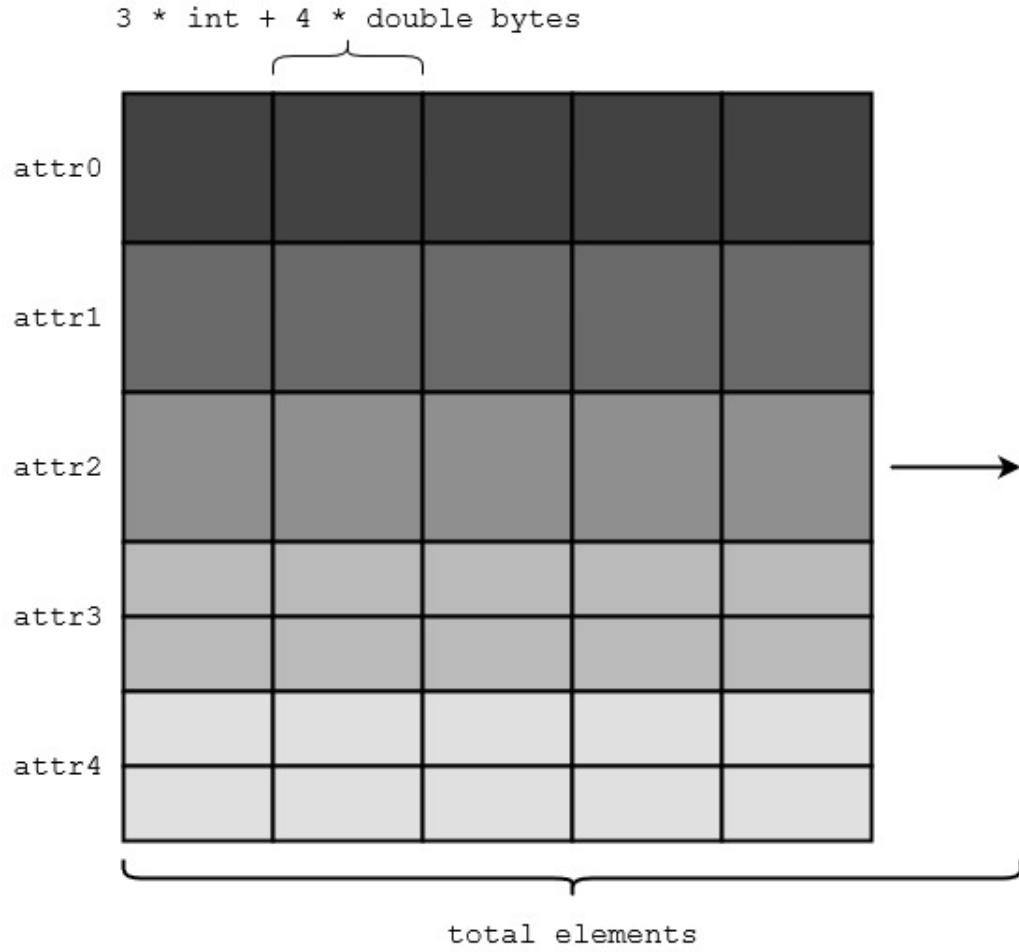


Figure 2.2: Beam Distributed Load Element layout in memory

```

1 type BeamDistributedLoads
2   integer(4)                                size;
3   integer(4), dimension(:)  , allocatable :: attr0;
4   integer(4), dimension(:)  , allocatable :: attr1;
5   integer(4), dimension(:)  , allocatable :: attr2;
6   real(8)    , dimension(:,:) , allocatable :: attr3;
7   real(8)    , dimension(:,:) , allocatable :: attr4;
8 end type
9 !attr3, attr4 second dimension has size 2

```

Figure 2.3: Memory format of Beam Distributed Load Elements in FORTRAN

```

1 function GetNumber(dbase, val0, val1, val2) result(cnt)
2 integer(4) cnt
3 type (BeamDistributedLoads) dbase
4 integer(4) val0, val1, val2
5 integer(4) i
6
7 cnt = 0;
8 do i = 1, dbase%size
9     if (dbase%attr0(i) == val0) then
10         if (dbase%attr1(i) == val1) then
11             if (dbase%attr2(i) == val2) then
12                 cnt = cnt + 1;
13             end if
14         end if
15     end if
16 end do
17
18 return
19 endfunction

```

Figure 2.4: The `GetNumber` function

As shown in detail in figure 2.4, the `GetNumber` function takes as input the `BeamDistributedLoads` Solver database, and three integer values `val0`, `val1`, `val2`. The output of `GetNumber` is the number of Beam Distributed Load Elements whose attributes `attr0`, `attr1`, `attr2` match the values `val0`, `val1`, `val2` one by one. `GetNumber` parses the `BeamDistributedLoads` Solver database in line 8 and compares each attribute to its corresponding input value in lines 9 - 11. If a set of `attr0`, `attr1`, `attr2` matches the input sequence described by `val0`, `val1`, `val2` then a counter variable is incremented in line 12. After the parsing is done, `GetNumber` returns this counter variable as declared in line 1.

The `GetAttributes` function takes the same input as the first one in addition to an integer value `count` representing a desirable counter value. The output of the function are the four floating-point values of the attributes `attr3`, `attr4` of the `BeamDistributedLoads` Solver database Element. The selection of this element is described as follows in figure 2.5. First a struct is declared that contains the return values of the function in lines 1 - 6. The `GetAttributes` function operates in the same way as the `GetNumber` function, in lines 15 - 20, by parsing the `BeamDistributedLoads` Solver database and incrementing a counter variable. In lines 21 - 23, if the counter variable reaches the values supplied by the input `count`, the index of the element is stored and the control flow breaks out of the loop. Lastly, in lines 29 - 32, the right Element is selected using the stored index and it's attributes `attr3`, `attr4` are returned.

```

1 type Attributes
2     real(8) v0;
3     real(8) v1;
4     real(8) v2;
5     real(8) v3;
6 end type
7
8 function GetAttributes(dbase, val0, val1, val2, count)
9     result(attrs)
10 type (Attributes) attrs
11 type (BeamDistributedLoads) dbase
12 integer(4) val0, val1, val2, count
13 integer(4) i, cnt, idx
14
15 cnt = 0;
16 do i = 1, dbase%size
17     if (dbase%attr0(i) == val0) then
18         if (dbase%attr1(i) == val1) then
19             if (dbase%attr2(i) == val2) then
20                 cnt = cnt + 1;
21                 if (cnt == count) then
22                     idx = i;
23                     exit;
24                 end if
25             end if
26         end if
27     end if
28 end do
29 attrs%v0 = dbase%attr3(idx,1);
30 attrs%v1 = dbase%attr3(idx,2);
31 attrs%v2 = dbase%attr4(idx,1);
32 attrs%v3 = dbase%attr4(idx,2);
33
34 return
35 endfunction

```

Figure 2.5: The GetAttributes function

```

1 integer(4) n, i
2 type(Attributes) attrs
3 ...
4 n = GetNumber(dbase, val0, val1, val2);
5 do i = 1, n
6     ...
7     attrs = GetAttributes(dbase, val0, val1, val2, n);
8     ...
9 end do
10 ...

```

Figure 2.6: Usage pattern of **GetNumber** and **GetAttributes** functions in the code base

2.2.4 Complexity Analysis of the iteration pattern

We pinpoint the reason why these two functions, and the iteration pattern in particular, pose a CPU bottleneck by

- performing an algorithmic complexity analysis on the iteration pattern
- examining the usage conventions of these functions in various parts of the code base

By examining the simplified code in figure 2.4 and figure 2.5 it is clear that the algorithmic complexity of the iteration pattern is $O(n)$, where n is the number of Beam Distributed Load Elements in the Solver database. The body of the iteration loop, i.e. an increment of the counter variable, is used as the unit of measurement. The number of Beam Distributed Load Elements in the Solver database, i.e. the n variable, is dependent on the Solver input file. By extension, it is clear that the algorithmic complexity of both **GetNumber** and **GetAttributes** is $O(n)$.

By examining the usage conventions of the two functions, with the help of the Visual Studio Profiler, we notice a usage pattern as described in figure 2.6, in many parts of the code base. First **GetNumber** is called and its return value is used as an iteration number limit. As part of the iteration body **GetAttributes** is called. Its return value is used in an irrelevant context. By calculating the total algorithmic complexity of the usage pattern we get a quadratic, $O(n^2)$ complexity. This is done by multiplying the **GetAttributes** complexity times the loop iterations and adding the **GetNumber** complexity, as follows

$$O(n) + n * O(n) = O(n) + O(n^2) = O(n^2)$$

It is clear that in many parts of the code, where this usage pattern is applied, a quadratic notation calculation is instantiated, presenting a clear bottleneck to the CPU.

2.2.5 The Memoization Model Requirements

The functions described above are part of a legacy code base. For this reason we cannot modify the control flow that leads to the call of these functions. To tackle this issue we provide a way to return the same result but in a reduced algorithmic complexity. The solution we propose is essentially a memoization mechanism to cache the return values of these functions and access them in a constant algorithmic complexity, using a key-value store. This solution comes with the following requirements

- The Beam Distributed Load Elements Solver database in memory needs to be immutable, after it is loaded. This way the *pure* property of the above functions is guaranteed. The property states that for a function to be *pure*, its return values are identical for identical arguments. By fulfilling the *pure* property we are able to create a mapping between a function input collection and its return value.
- The collection of all the `BeamDistributedLoads` Element attribute permutations, `attr0`, `attr1`, `attr2` must not be unreasonably large. Each attribute permutation corresponds to a mapping in the key-value store. This ensures that the memory footprint of our solution will stay in a reasonable limit.

Regarding the first requirement, we know that the collection of structural elements of the model is indeed immutable, after consulting the architects of RAF. That is, after the Solver loads the structural elements of the input file, they take the form of read-only attributes in memory, while the analysis step takes place.

Regarding the second requirement, we run an analysis of our own on a large size input file, to determine the size of attribute permutations. We produce the following results. `attr0` can have a value in the range of $[1, 5220]$. `attr1` can have a value in the range of $[1, 2]$. `attr2` can have only the value of 1. Knowing that each permutation takes up 12 bytes in memory, since a permutation is comprised of three 4-byte integer numbers `attr0`, `attr1`, `attr2`, we can approximate the memory footprint that comes with storing the total number of permutations as follows

$$\begin{aligned}
 \text{permutation size} &= (\text{attr0 value range size}) \\
 &\quad * (\text{attr1 value range size}) \\
 &\quad * (\text{attr2 value range size}) \\
 &\quad * (\text{permutation tuple size}) \\
 &\quad * (\text{size of integer in bytes}) &\Rightarrow \\
 \text{permutation size} &= 5220 * 2 * 1 * 3 * 4 &\Rightarrow \\
 \text{permutation size} &= 125280 \text{ bytes} \approx 122 \text{ kilobytes}
 \end{aligned}$$

```

28 integer(4) function CachedLoadsGetCount(structured_type, ielem, icase, iload)
29     !DEC$ ATTRIBUTES STDCALL, ALIAS: 'CachedLoadsGetCount' :: CachedLoadsGetCount
30     !DEC$ ATTRIBUTES REFERENCE::structured_type;
31     !DEC$ ATTRIBUTES REFERENCE::ielem;
32     !DEC$ ATTRIBUTES REFERENCE::icase;
33     !DEC$ ATTRIBUTES REFERENCE::iload;
34 implicit none;
35 integer(4), intent(IN) :: structured_type
36 integer(4), intent(IN) :: ielem
37 integer(4), intent(IN) :: icase
38 integer(4), intent(IN) :: iload
39 endfunction CachedLoadsGetCount

```

Figure 2.7: FORTRAN to C interoperability function signature

We confirm the legibility of our results with the architects of RAF. These results show us that the memory footprint of our model is of a reasonable size.

Before implementing our solutions we ensure we can take an extra step of future proofing them. Most of the Solver code base is written in FORTRAN. During the time span of our work, the architects of RAF are in the process of migrating the Solver code base to the C++ language. We wanted to make sure that the implementation of our solution would not need a future code migration. Since the bottleneck has been found in a FORTRAN part of the code base, we implement our solution in C++ and then use FORTRAN to C interoperability routines to inform the compiler to link C++ with FORTRAN object files together. In the future, one would only need to eliminate these interoperability function calls to port our solution in C++.

2.2.6 The Memoization Model Interface

To tackle the bottleneck problem we create a mapping between a permutation of `attr1`, `attr2`, `attr3` and the number of `BeamDistributedLoads` Elements that contain this permutation. We store our mappings using a key-value store. We provide two different C++ implementations of the key-value store. One using a Hash Table and another one using a Trie data structure.

Both our solutions share a common interface. We also give the architects of RAF the option to switch between a preferred implementation at compile time.

Our interface consists of a private state object and three different public operations in the form of functions.

The state object emulates the key-value store depending on the selected implementation. From now on we refer to the key-value store as the *CachedLoadsMap*. The *CachedLoadsMap* holds mappings between unique keys and values. A *CachedLoadsMap* key is a permutation of values of `BeamDistributedLoads` Element attributes `attr1`, `attr2`, `attr3`.

As shown in figure 2.8, a *CachedLoadsMap* value contains the number of `BeamDistributedLoads` Elements whose attributes match the permutation of the

```

1 struct Value {
2     int count;
3     std::vector<int> indices;
4 };

```

Figure 2.8: The value signature of a mapping of *CachedLoadsMap*

```

1 void Insert (const BeamDistributedLoads& dbase);
2 int  GetCount(int val0, int val1, int val2);
3 int  GetIndex(int val0, int val1, int val2, int position);

```

Figure 2.9: The interface signatures

CachedLoadsMap key, we refer to this number as *count*. A *CachedLoadsMap* value also contains a collection of indices of those Elements as indexed in the *BeamDistributedLoads* Solver database, we refer to this collection as *indices*. Each index corresponds to an element whose attributes match the permutation of its corresponding key. The *indices* are stores in ascending order, while their size matches their respective *count*.

As shown by their signatures in figure 2.9. The three public operations are *Insert*, *GetCount*, *GetIndex*.

Insert is used to initialize the *CachedLoadsMap*. The input of *Insert* is the *BeamDistributedLoads* Solver database. We populate the *CachedLoadsMap* based on the element attributes. We parse the database fields inserting all possible attributes, *attr0*, *attr1*, *attr2*, to *CachedLoadsMap*. During parsing, if a permutation is found more than once, we update the value of the associated mapping.

GetCount is used to eliminate the linear complexity of the *GetNumber* function. It takes as input a permutation of attribute values *val0*, *val1*, *val2* as key, accesses *CachedLoadsMap* and returns the *count* field of the mapped value.

GetIndex is used to eliminate the linear complexity of the *GetAttributes* function. It takes as input a permutation of attribute values *val0*, *val1*, *val2* as key as well as a *position* parameter, accesses *CachedLoadsMap*, accesses the mapped value's *indices* collection and returns the index at *position*.

The detailed functionality as well as the algorithmic complexity of these operations depends on their respective implementation.

2.2.7 The Memoization Model Hash Table Implementation

One of our implementations of the interface uses a Hash Table as the *CachedLoadsMap* object, as shown in figure 2.10 line 9. A *std::unordered_map* container of the STL library is essentially our *CachedLoadsMap* instance. *std::unordered_map* is an associative container that contains key-value pairs with unique keys. Internally,

```

1 struct Key {
2     int val0;
3     int val1;
4     int val2;
5 };
6
7 class UnorderedMapImpl {
8 private:
9     std::unordered_map<Key, Value> map{};
10
11 public:
12     void Insert(const Key& key, int index);
13     auto Find(const Key& key) -> Value;
14 };
15
16 void UnorderedMapImpl::Insert(const Key& key, int index) {
17     auto it = map.find(key);
18     if (it == std::cend(map)) {
19         const auto& [newIt, _] = map.emplace(key, Value{});
20         it = newIt;
21     }
22     it->second.count += 1;
23     it->second.indices.emplace_back(index);
24 }
25
26 auto UnorderedMapImpl::Find(const Key& key) -> Value {
27     auto it = map.find(key);
28     return (it != std::cend(map) ? it->second : Value{});
29 }

```

Figure 2.10: The UnorderedMapImpl class

the elements are not sorted in any particular order, but organized into buckets. Which bucket an element is placed into depends entirely on the hash of its key. Keys with the same hash code appear in the same bucket [33].

The signature of our key is a struct of three integer values to hold a permutation of attributes `attr0`, `attr1`, `attr2`, as shown in figure 2.10 lines 2 - 4. The signature of our value is the aforementioned struct described in figure 2.8. The private interface of the implementation supports two operations, **Insert** and **Find**.

The implementation of **Insert** is shown in figure 2.10 from line 16. The two input arguments of **Insert** are the key to be inserted or updated and the index of the `BeamDistributedLoads` Solver database that is associated with the permutation of `val0`, `val1`, `val2` described by the key. In lines 17 - 21 we search for the key input in `CachedLoadsMap` and insert it, if a mapping does not already exist. In lines 22 - 23 we construct or update the associated value by increasing its `count` field by one and by appending the index to the end of its `indices` collection.

The implementation of **Find** is shown in figure 2.10 from line 26. The input arguments of **Find** are the key to be searched. The return value of **Find** is the value associated with the input key in `CachedLoadsMap`. If the input does not exist in `CachedLoadsMap`, an empty value is returned. However, for the use cases of our work, we always supply a valid key. **Find** is utilized by both **GetCount** and **GetIndex** public interface operations. **GetCount** returns the `count` field of the associated value returned by **Find**. **GetIndex** accesses the value's `indices` collection and returns the element found at `position`.

Throughout C++ code *emplace* semantics are used when inserting new elements to STL containers to construct these elements in place. This way we avoid any extra copy or move operations required when using standard *insert* or *push_back* semantics.

The algorithmic complexity of both **Insert** and **Find** is $O(1)$, since both insert and look-up operations to a Hash Table take constant time on average as stated in the C++ reference pages [35], [34].

2.2.8 The Memoization Model Trie Implementation

One of our implementations of the interface uses a Trie as the `CachedLoadsMap` object, as shown in figure 2.11 line 14. A Trie is a tree data structure used to locate keys within a set. In most cases these keys are strings comprised of multiple linked nodes, where each node contains an individual character of the key. In our case, the key is a permutation of attributes `attr0`, `attr1`, `attr2` while each node consists of an attribute value that is part of a permutation. This way, we form a tree data structure where the children of an attribute value is the collection of all possible attribute values that can come next, forming permutation paths. The leaves of the tree contain valid contents of a value as shown in figure 2.8. A visualization of a Trie is shown in figure 2.13. The private interface of the implementation supports two operations, **Insert** and **Find**.

The implementation of **Insert** is shown in figure 2.12 from line 1. The two

```

1 class TrieImpl {
2 private:
3     struct Node;
4 public:
5     using NodeUPtr = std::unique_ptr<Node>;
6     using ChildrenMap = std::unordered_map<int, NodeUPtr>;
7     using Key = std::vector<int>;
8
9 private:
10    struct Node {
11        ChildrenMap children{};
12        Value value{};
13    };
14    NodeUPtr root{std::make_unique<Node>()};
15
16 public:
17    void Insert(const Key& path, const int& index);
18    auto Find(const Key& path) -> Value;
19 };

```

Figure 2.11: The `TrieImpl` class signature

input arguments of `Insert` are the key to be inserted or updated and the index of the `BeamDistributedLoads` Solver database that is associated with the permutation of `val0`, `val1`, `val2` described by the key. In lines 3 - 9 we follow the permutation path described by the input key, inserting any nodes that are missing along the way. Once we have reached a leaf node, we construct or update the contents of the value, in lines 10 - 11.

The implementation of `Find` is shown in figure 2.12 from line 14. The input arguments of `Find` are the key to be searched. The return value of `Find` is the value associated with the input key in `CachedLoadsMap`. In lines 16 - 21 we follow the input key path until we reach a leaf node. In line 22 we return the value of the leaf node. If we do not manage to completely follow the path described by the input key until the end, then the key not exist, so we return an empty value, in line 19. However, for the use cases of our work, we always supply a valid key. `Find` is utilized by both `GetCount` and `GetIndex` public interface operations. `GetCount` returns the `count` field of the associated value returned by `Find`. `GetIndex` accesses the value's `indices` collection and returns the element found at `position`.

Throughout C++ code `std::unique_ptr` STL semantics are used for the pointer attribute fields of the `TrieImpl` class. `std::unique_ptr` is a smart pointer that owns and manages another object through a pointer and disposes of that object when the `unique_ptr` goes out of scope [32]. Use of `std::unique_ptr` instances help us denote the ownership of objects in a syntactic way, as opposed to using raw pointers,

```
1 void TrieImpl::Insert(const Key& key, int index) {
2     Node* node = root.get();
3     for (const auto& n : key) {
4         if (node->children.find(n) ==
5             std::cend(node->children))
6             node->children.emplace(
7                 n, std::make_unique<Node>());
8         node = node->children.at(n).get();
9     }
10    node->value.count += 1;
11    node->value.indices.emplace_back(index);
12 }
13
14 auto TrieImpl::Find(const Key& key) -> Value {
15     Node* node = root.get();
16     for (const auto& n : key) {
17         if (node->children.find(n) ==
18             std::cend(node->children))
19             return Value{};
20         node = node->children.at(n).get();
21     }
22     return node->value;
23 }
```

Figure 2.12: The `TrieImpl` class methods

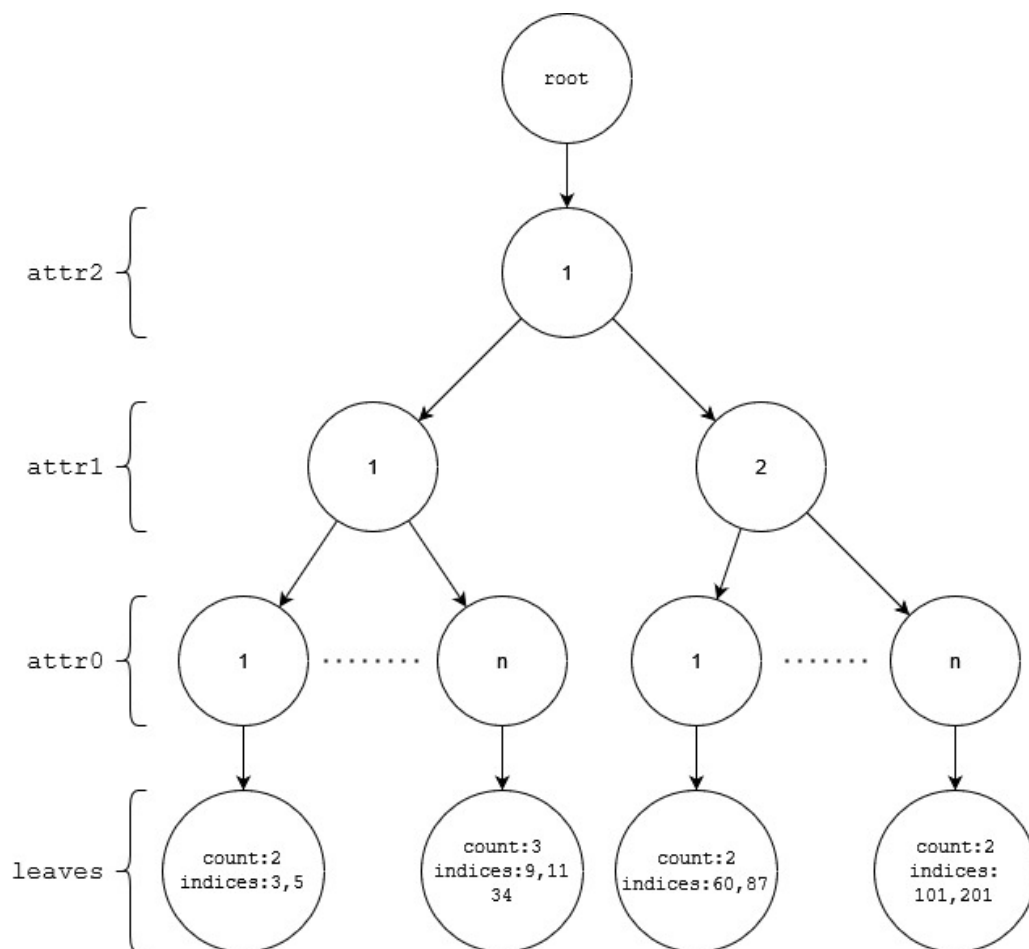


Figure 2.13: Optimally constructed Trie in terms of node size

where ownership is not communicated to the user. `std::unique_ptr` instances also relieve us of the mental task of recalling to delete dynamically allocated memory once the lifetime of an object has ended. `std::unique_ptr` handles the deallocation of memory, once the owned object goes out of scope.

The algorithmic complexity of both **Insert** and **Find** is $O(1)$. The complexity of accessing a Trie is linear to the length of the key, since we traverse a tree path with number on nodes equal to the total elements the key is comprised of. In our case, the key length is 3 items, evaluating to a constant time complexity. To store pointers to the children of each node we use a `std::unordered_map` STL container. Access complexity to this container is constant time, so it does not affect the total access complexity of the Trie.

To construct a Trie with the minimum number of nodes, we need to take into account the value range of each part of the key and the depth of the tree with which it is associated. By consulting our analysis to determine the size of attribute

permutations in section 2.2.5 as well as the architects of RAF we know that `attr0` can have a value in a range that can span to large number. `attr1` can have a value in the range of $[1, 2]$. `attr2` can have only the value of 1. To construct our Trie having wit the minimum number of nodes, we place the attribute with the highest value range closest to the root. Then we place the rest of the attributes in a descending order according their value ranges, as we go down the levels of the tree. Figure 2.13 demonstrates an optimally constructed Trie.

2.2.9 Linking our work with the code base

To link our work with the pre-existing Solver code base we follow the steps below

- Design and tweak our interface in way that allows extensibility with other structural elements, apart from Beam Distributed Load Elements, while we maintain its ability to be linked with pre-existing code.
- Encapsulate our interface public C++ operations, into FORTRAN function calls using FORTRAN to C interoperability semantics.
- Replace the problematic iteration patterns in FORTRAN code with the above FORTRAN-encapsulated interface operations.

The problematic iteration pattern was not only applied in parts of the code that had to do with Beam Distributed Load Elements. After examining the results of our profiling session we notice the same pattern appearing on code segments related to Concentrated Force Load Elements as well as similar patterns having been applied for other elements. After completing our Memoization Model implementation regarding Beam Distributed Load Elements, we extend the interface in order to include Concentrated Force Load Elements as well. We achieve this by extending the key signature of our *CachedLoadsMap* container to additionally include the type of the element i.e. Beam Distributed Load, Concentrated Force Load. This way, we maintain the basic structure of our interface while enabling the addition of similar structural elements.

To enable calling our interface functions using FORTRAN code we encapsulate each of our public operations into a a FORTRAN function with the exact same signature. We provide these signatures with any interoperability language syntax needed to successfully compile and link the object files together.

Lastly, we replace the problematic iteration patterns in legacy code with calls to the above functions, as well as make any other necessary adjustments in code as shown in figure 2.14. We do not forget to call the Insert operation in the part of the code after the Solver database has been generated.

2.3 Evaluation

We evaluate our work in terms of regression testing and performance profiling. To perform our testing we supply the Solver with a collection of input files. For

```

1 function GetNumber(dbase, val0, val1, val2) result(cnt)
2 integer(4) cnt
3 type (BeamDistributedLoads) dbase
4 integer(4) val0, val1, val2
5
6 cnt = GetCount(val0, val1, val2);
7
8 return
9 endfunction
10
11 function GetAttributes(dbase, val0, val1, val2, count)
12     result(attrs)
13 type(Attributes) attrs
14 type (BeamDistributedLoads) dbase
15 integer(4) val0, val1, val2, count
16 integer(4) idx
17
18 idx = GetIndex(val0, val1, val2, count);
19 attrs%v0 = dbase%attr3(idx,1);
20 attrs%v1 = dbase%attr3(idx,2);
21 attrs%v2 = dbase%attr4(idx,1);
22 attrs%v3 = dbase%attr4(idx,2);
23
24 return
25 endfunction

```

Figure 2.14: The new `GetNumber` and `GetAttributes` functions.

regression testing, we compare the Solver analysis results of each input file. For performance profiling, we benchmark the total execution time of the Solver.

2.3.1 The input file collection

The Solver takes as input a structural model in the form an .XML file that adheres to a specific .XSD schema. After the Solver performs structural analysis on the input file, an output .ROU file is produced, containing the analysis results. We have collected a number of input files which help us test the Solver.

Our input collection consists of the following files

- *u_dist48000.xml*
- *u_dist16170.xml*
- *g_dist55000.xml*
- *g_conc55000.xml*
- *g_dist55000_conc55000.xml*
- *g_dist29800_conc29800.xml*
- *g_dist10200_conc10200.xml*

u_dist48000.xml, *u_dist16170.xml* are end-user generated structural models. We use end-user generated structural models to evaluate our work based on real-life data. This way, the performance evaluation simulates end-user times more closely. *u_dist48000* contains 48000 Beam Distributed Load Elements while *u_dist16170* contains 16170. The higher the number of Beam Distributed Load Elements or Beam Concentrated Force Load Elements in an input file, the more times our work is evaluated during execution.

g_dist55000.xml, *g_conc55000.xml*, *g_dist55000_conc55000.xml*, *g_dist29800_conc29800.xml*, *g_dist10200_conc10200.xml* are tool generated input files containing Beam Distributed Load Elements and Beam Concentrated Force Load Elements. *g_dist55000.xml* and *g_conc55000.xml* contain 55000 Beam Distributed Load Elements and 55000 Beam Concentrated Force Load Elements respectively. *g_dist55000_conc55000.xml* contains 55000 Beam Distributed Load Elements as well as 55000 Beam Concentrated Force Load Elements. Same pattern goes for *g_dist29800_conc29800.xml* and *g_dist10200_conc10200.xml*.

Due to the lack of end-user structural models, we implement a Solver Input Generator tool in Python to generate more input files for testing.

2.3.2 Contents of an input file

An input file represents a structural model comprised of various structural elements. Some of these elements are the following

- *Storey*, a property that represents the floor of a building.
- *Node*, a structural component that represents a connection point among components like *Beams*. A *Node* is tied to a storey.
- *Beam*, a horizontal or vertical structural component that represents a connection between two *Nodes*.
- *Shell*, a two dimensional plane component that represents a surface. It is a connection among three of four *Nodes*.
- *Restrain*, a *Node* property that represents the *Node*'s movement and rotation restrictions among the *xyz* axes.
- *Node Mass*, a *Node* property that represents the *Node*'s mass.
- *Node Force*, a *Node* property that represents an external force applied to the *Node*.
- *Beam Distributed Load*, a *Beam* property that represents an external force applied evenly along the *Beam*'s length.
- *Concentrated Force Load*, a *Beam* property that represents an external force applied to a specific point along the *Beam*'s length.

Our work is applied on the Solver memory database of Beam Distributed Load Elements and Beam Concentrated Force Load Elements. A high number of these elements in an input file leads the Solver execution control flow to take advantage of our Memoization Model multiple times.

2.3.3 The input file generator script

Due to the lack of more end-user structural models, we implement a tool in Python that helps us generate and visualize Solver input files. This tool takes as input the three dimensional size of a structural model in the form of 3 integers and produces a fully connected structural model, while applying multiple Distributed Loads or Concentrated Force Loads per *Beam*. We use a template .XML input file and .XSD schema and extend their contents by injecting XML elements. Furthermore, we use a graphics library to visualize the generated model as shown in figure 2.15. Green lines represent *Beam* connections while red squares represent *Nodes*.

2.3.4 Regression Testing

We need to make sure that the integrity of the structural analysis results of the Solver is not invalidated. We conduct regression testing using our input files collection. We use an input file to execute the Solver before applying our Memoization Model. Then we use the same input file to execute the Solver after applying our

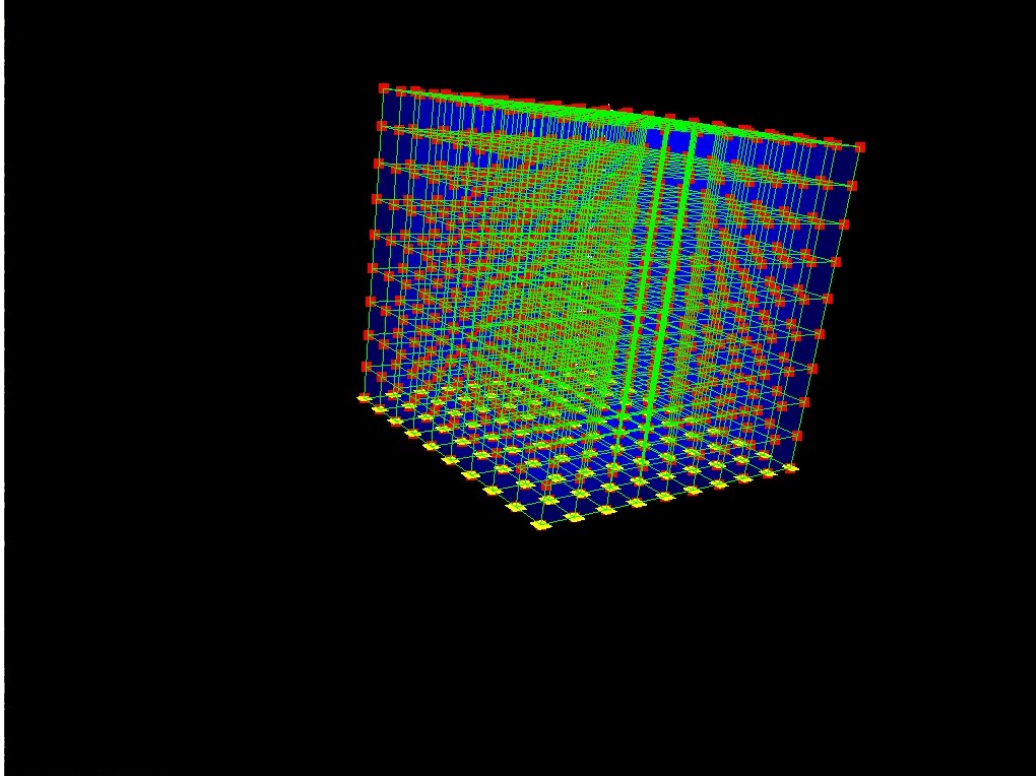


Figure 2.15: A 10 x 10 x 10 generated structural model

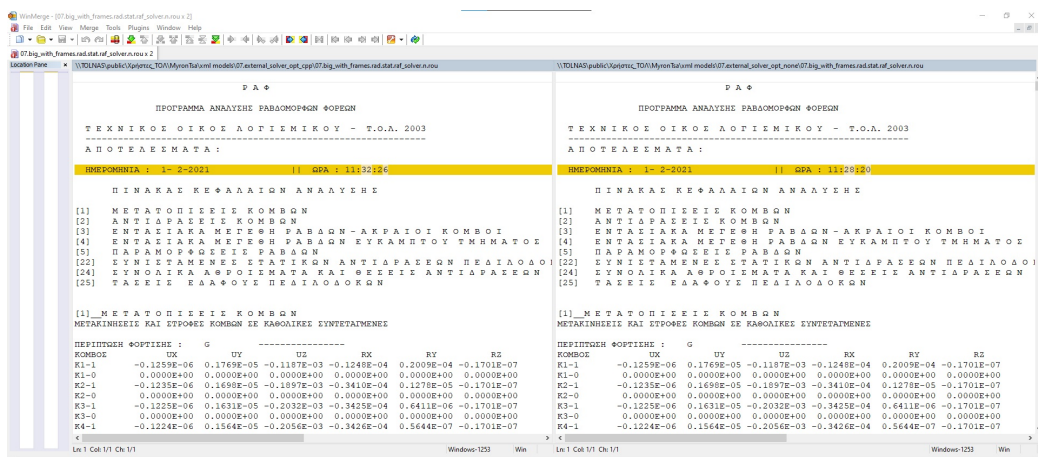


Figure 2.16: Comparing two versions of the same output file

Memoization Model. Then we compare the respective output files using a file content comparison tool, as shown in figure 2.16. If both files are identical, then regression testing is a success. We perform regression testing to all the implementations of the Memoization Model.

2.3.5 Performance Evaluation

To observe the performance benefits of the Memoization Model we need to profile it in terms of execution time and *speedup*. We conduct performance profiling using our input file collection and by timing the total execution time of the Solver.

We conduct performance profiling using all the files of our input collection. We profile the total execution of the Solver on each file by inserting timepoints at the beginning and end of the execution of the Solver and by calculating the total duration. We disable output file generation during profiling.

We test each input file on the following cases

- Without using the Memoization Model.
- Using the Hash Table implementation of the Memoization Model.
- Using the Trie implementation of the Memoization Model.
- Using a FORTRAN implementation of the Memoization Model.

The last implementation is provided by the architects of RAF and uses raw FORTRAN arrays to store data.

While presenting our performance results we use the concept of *speedup* as introduced in the book *Computer architecture : a quantitative approach*, on pages 46 - 47 [26].

$$speedup = \frac{T_{old}}{T_{new}}$$

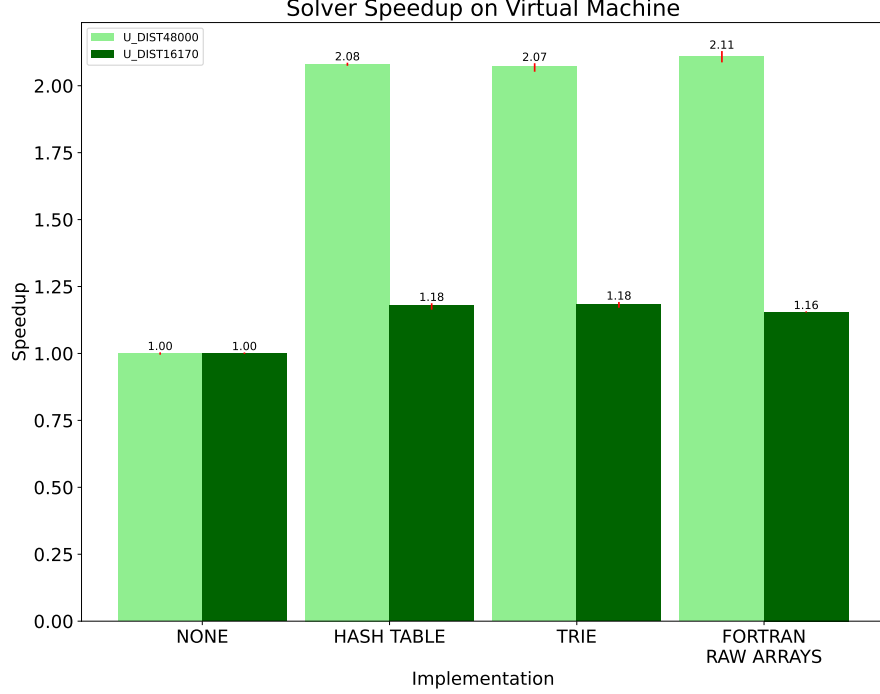
where T_{old} is the execution time of the Solver on a specific input file without using any implementation of the Memoization Model and T_{new} is the execution time of the Solver on the same input file using an implementation of the Memoization Model.

We conduct our performance profiling on two different machines. One of them is a Virtual Machine set up on a NUMA architecture, while the other one is a typical end-user PC.

2.3.6 Evaluation on NUMA architecture Virtual Machine

The specifications of our machine are the following.

- CPU, Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz. Contains 2 NUMA nodes of 14 physical cores each. Each of the 28 physical cores supports hyper-threading technology totalling to 56 logical cores.

Figure 2.17: Memoization Model *speedup* on all user input files

- RAM, 16 GB at 1500MHz
- OS, Microsoft Windows 10 Pro 64-bit

We repeat each measurement 3 times and present an average value.

In figure 2.17 we present the average *speedup* of each of our implementations on different end-user input files. The horizontal axis shows our **UnionManager** implementations while the vertical axis shows the *speedup* when compared with the solver design without memoization. Each bar color represents a different input file. The red error bars represent the standard deviation of each measurement. We observe an approximate *speedup* of 2.0 regarding the input file with 48000 Distributed Load Elements. On the input file of 161700 elements the *speedup* is approximately 1.2 due to the solver spending most of the execution time on operations not related to our work.

In figure 2.18 we present the average *speedup* of each of our implementations on different generated input files. The horizontal axis shows our **UnionManager** implementations while the vertical axis shows the *speedup* when compared with the solver design without memoization. Each bar color represents a different input file. The red error bars represent the standard deviation of each measurement. Regarding the blue colored inputs, which contain elements of a single category

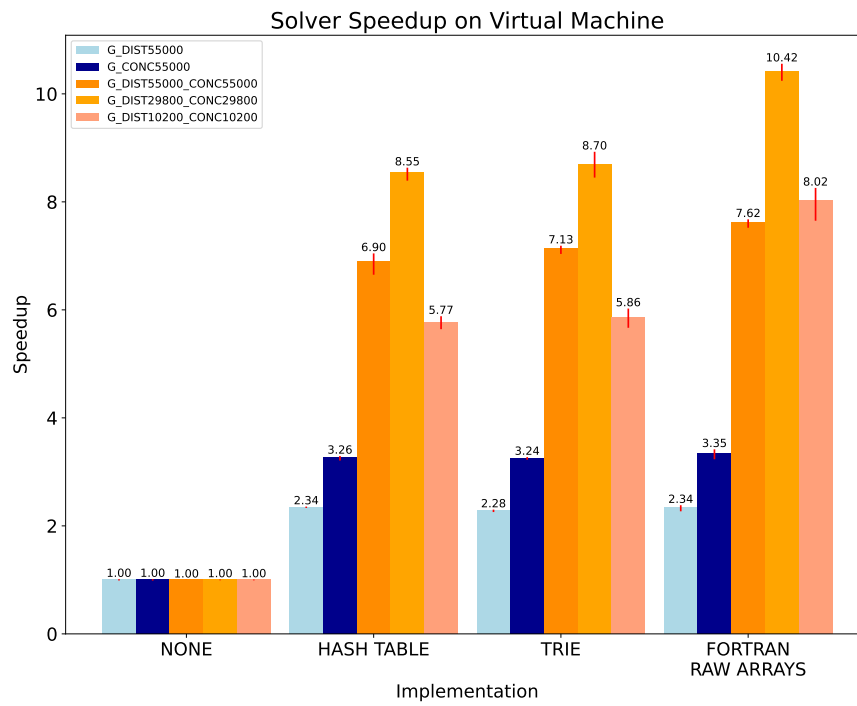


Figure 2.18: Memoization Model *speedup* on all generated input files

each, we observe an approximate speedup of 2.3 and 3.3 respectively, among all implementations. Regarding the orange colored inputs, which contain elements of two categories each, we observe an approximate speedup of 7, 8.6 and 5.8 respectively among the *HASH_TABLE* and *TRIE* implementations. The *FORTRAN_RAW_ARRAYS* implementation offers a higher speedup due to the absence of hashing. Generated input files offer a higher approximate speedup than user input files due the solver spending most of its execution time on memoized operations.

2.3.7 Evaluation on the user PC machine

The specifications of our machine are the following.

- CPU, Intel Core i7-4790S CPU at 3.20 GHz, 4 cores, 8 Logical Processors due to hyper-threading technology
- RAM, 16 GB at 1500MHz
- SSD, Samsung SSD 860 EVO 250GB
- OS, Microsoft Windows 10 Home 64-bit

Due to time constraints we do not repeat any measurements on our evaluation on the user PC machine.

In figure 2.19 we present the *speedup* of each of our implementations on the end-user generated structural model input file *u_dist48000.xml*. The vertical axis shows our **UnionManager** implementations while the horizontal axis shows total the execution time. We observe a difference of approximately 100 seconds from the original execution when applying any of our implementations, leading to a about 2.0 *speedup* factor. The time spent on the problematic iteration pattern, as denoted by our performance profiling analysis in section 2.2.1, has now disappeared, allowing the linear algebra calculations of the solving process to take up the majority of the execution time.

In figure 2.20 we present the execution time of each of our implementations on the tool generated input file *g_dist55000_conc55000.xml*. The vertical axis shows our **UnionManager** implementations while the horizontal axis shows total the execution time. We observe a difference of approximately 280 seconds from the original execution when applying any of our implementations, leading to a about 6.6 *speedup* factor. When we compare *g_dist55000_conc55000.xml* with *u_dist48000.xml* we notice that, in *g_dist55000_conc55000.xml*, the ratio of the sum of Beam Distributed Load Elements and Beam Concentrated Force Load Elements to the sum of the other structural elements is particularly high. Since we apply our work on operations regarding Beam Distributed Load Elements and Beam Concentrated Force Load Elements, the large *speedup* ratio on *g_dist55000_conc55000.xml* is an expected result.

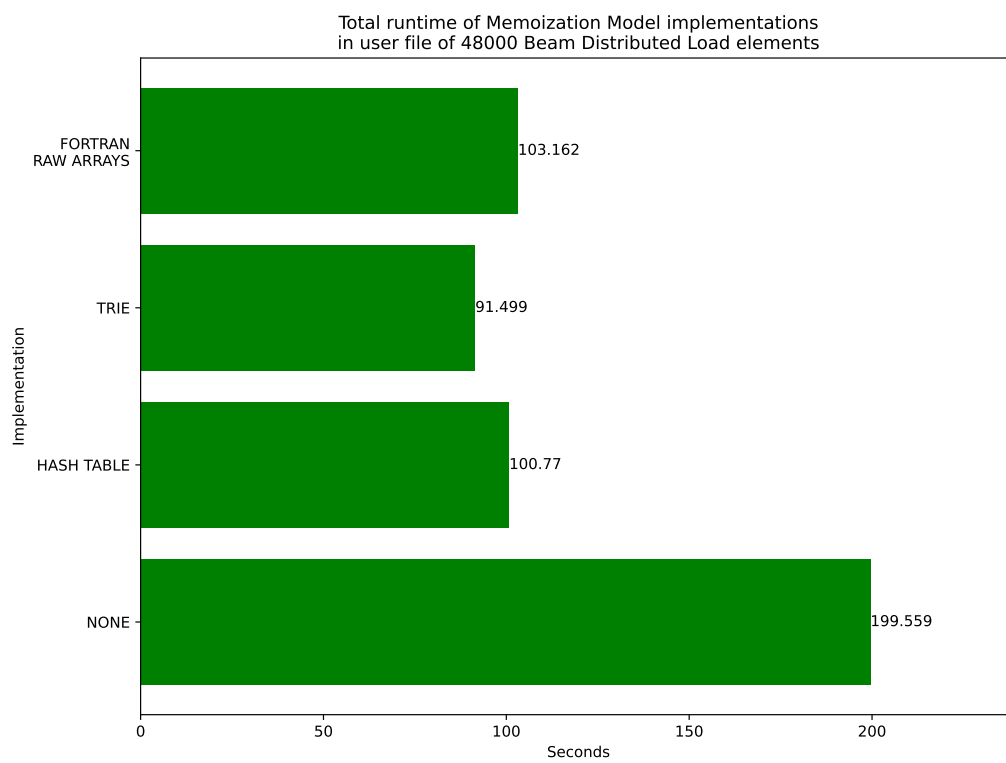


Figure 2.19: Memoization Model execution time on user input

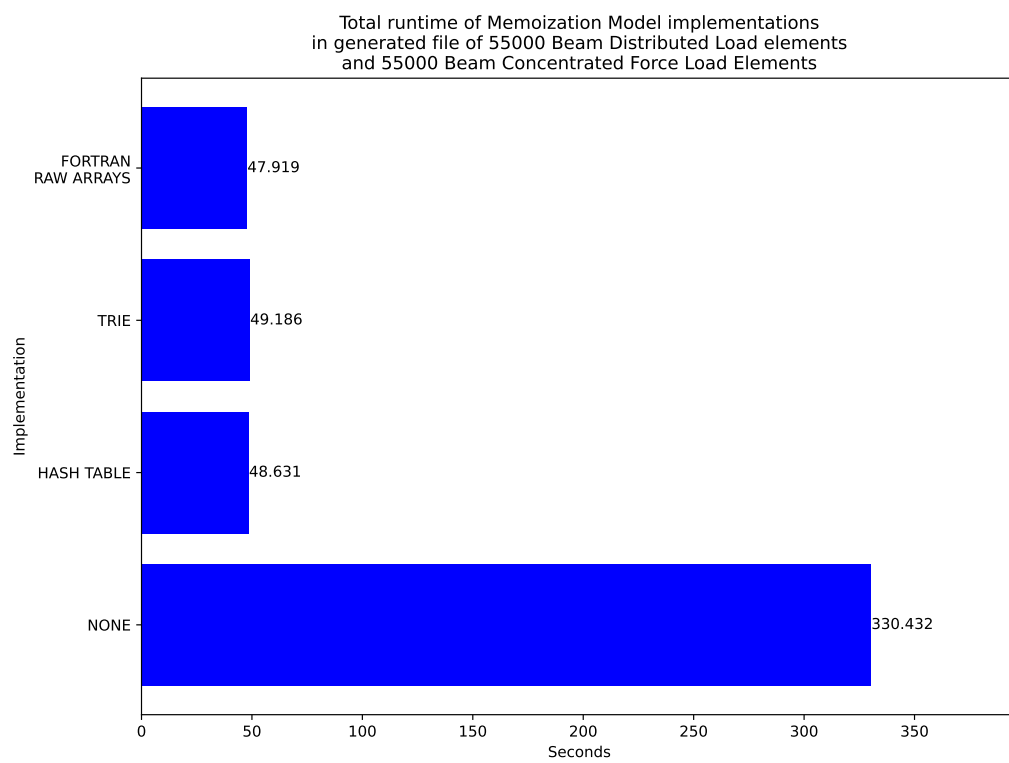
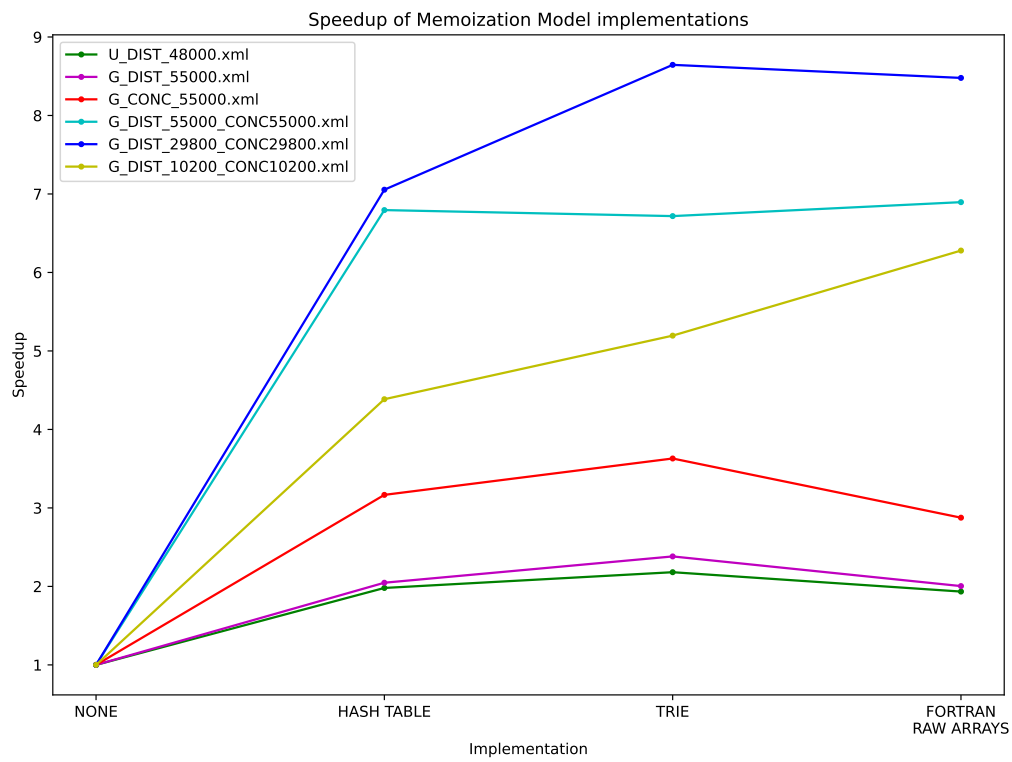


Figure 2.20: Memoization Model execution time on generated input

Figure 2.21: Memoization Model *speedup* on all inputs

In figure 2.21 we present the *speedup* of each of our implementations on multiple input files. The horizontal axis shows our `OnionManager` implementations while the vertical axis shows the *speedup*.

2.4 Related Work

The idea of a Trie data structure was described in 1960 by *Edward Fredkin*, who also coined the term *Trie*. In the article *Trie memory* [7], he describes several paradigms of trie memory and compares them to other memory paradigms.

Many Memoization techniques, which present similarities to our own work and methodology, have been presented in the past.

Donald Michie in *"Memo" Functions and Machine Learning* [18] coined the term Memoization to describe the process where a function can "remember" previously computed results.

R.S. Bird in *Tabulation Techniques for Recursive Programs* [4] describes "a process in which function values are computed once only and then stored in some conveniently represented table for future use. Subsequent requests for the value are answered by looking up the appropriate table entry.

John Hughes in *Lazy-memo-functions* [11] suggests applying memory address equality between the current arguments of a function call and the memoized arguments in order to reduce the equality checking costs among compound data-structures.

Yanhong A. Liu, Scott D. Stoller, Tim Teitelbaum present the *cache-and-prune* method in *Static Caching for Incremental Computation* [16]. This method presents an way to "incrementalize a program in order to use cached results of intermediate computations". The first two steps of the method are similar to the methodology of our own work.

Mostow and Cohen in *Automating Program Speedup by Deciding What to Cache* [21] present an extended analysis on common thought processes of optimization strategies. Deciding what and when to cache as well as side effects of caching, such as state changes are explored in their work.

Helmut A. Partsch in *Specification and transformation of programs: a formal approach to software development* [25] discusses the application of the techniques of Memoization and Tabulation, also known as dynamic programming, to the Fibonacci function.

Umut A. Acar, Guy E. Blelloch, Robert Harper in *Selective memoization* [1] present a framework for applying memoization selectively, allowing the user to fine-grain the application of memoization according to the application needs.

2.5 Future Work

In our work we present a model to memoize the return values of specific functions. We focus on functions related to Beam Distributed Load Elements. However, we

have extended the signature of our attribute to key to additionally include the type of the element whose value we memoize. This made it possible for us to extend the functionality of our model to functions related to Beam Concentrated Force Load Elements. This way, our model can be extended to memoize return values of functions related to other structural elements as well, provided that these elements have a similar data format.

Our work can also be extended by providing a way to prune the memoized values that are not used in any way. During the initialization step of our model, a mechanism to filter out attribute keys can be applied, storing only the return values that correspond to attributes keys that are used by the Solver. However, this mechanism requires a way to determine which keys are useful. While there is no definite way to answer that, analyzing many user files to gather key instances that are repeated throughout each file, is a starting step to the right direction.

Additionally, our model could be restructured and applied in a more classic memoization context. Instead of an initialization step, in which we memoize all the possible return value, a value can be memoized when calling the function with specific arguments for the first time. Then on subsequent times, the memoized value that matches the arguments will be retrieved. While, this adds a one-time computational overhead at the first time the function is called, it simplifies the code as well as renders the initialization step obsolete.

Lastly, the possibility of evaluating the performance of our work on more end-user generated files remains open. This would help us get more accurate results regarding user functionality, since the amount of end-user files available to us, during the time of our work, is limited.

Chapter 3

Capacity Volume Manager Concurrency

3.1 Background

The Capacity Volume Manager is another RAF module. Its code is integrated to the main RAF application along with its own UI windows and buttons. The main RAF application is developed in C++ using the Visual Studio IDE [20]. The Capacity Volume Manager is responsible for storing, calculating and providing the user with the requested Capacity Volumes. A Capacity volume is an attribute of a structural element. It contains information about the resistance of the element to winds, earthquakes and other external forces. A Capacity Volume can also be represented graphically, while its calculation is a time consuming, CPU intensive operation. Capacity Volume Calculation is based on the method described by *Werner H.* in the article *Inclined bending of polygonal bordered reinforced concrete cross-sections* [36].

The end-user can request the Capacity Volume of an element. The request is delegated to the Capacity Volume Manager. If the Capacity Volume is already calculated, the Manager supplies it to the rendering engine and a graphical representation of the Capacity Volume is shown on screen. If it is not already calculated, the Manager calculates and stores it before proceeding with the rest of the operations.

The end-user can also request the Capacity Volumes of multiple elements. These are handled in and calculated as a batch by the Capacity Volume Manager following the above procedure in a linear fashion. During calculation, a progress bar is shown on screen to inform the user of the completion status of the operation.

We have been informed by the architects of RAF that Capacity Volumes are data unrelated entities in order to research the possibility of simultaneous calculation by utilizing more than one CPU cores.

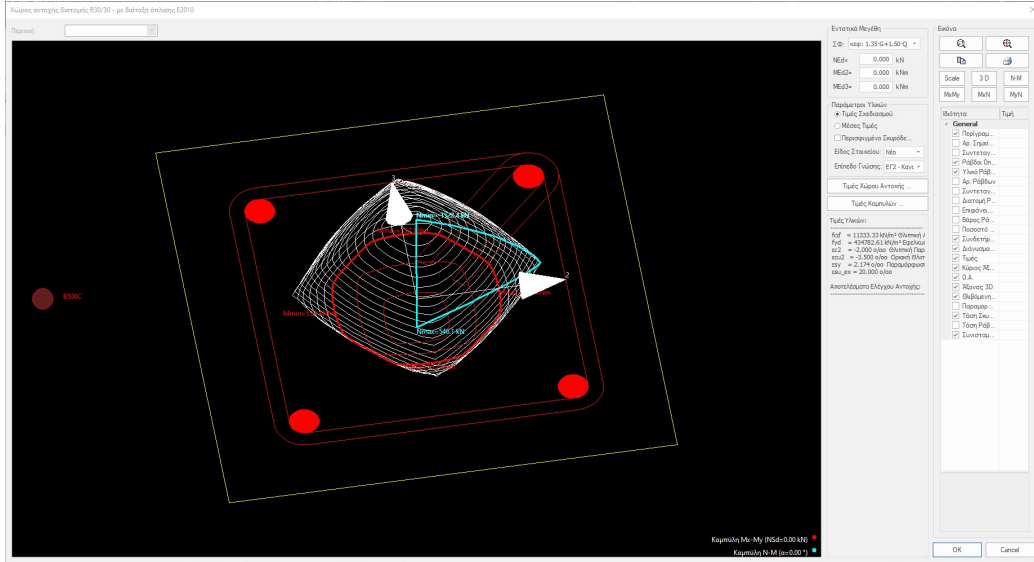


Figure 3.1: Graphical representation of Capacity Volume

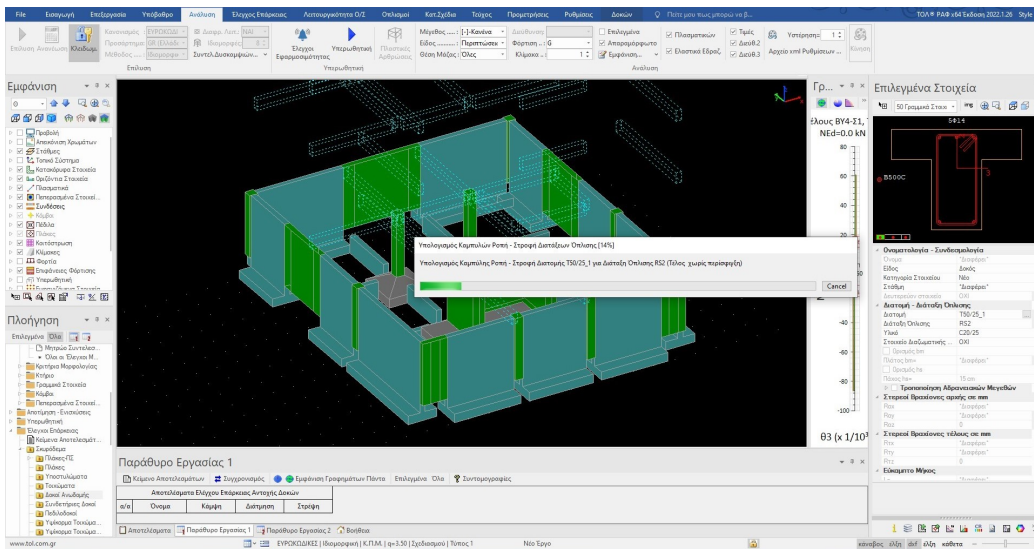


Figure 3.2: Capacity Volume Calculation Progress bar

```

1 enum MatProps { ... }; // Material Properties type
2 class ReinforceSchema { ... }; // Reinforce Schema type
3 using ReinforceSchemaSPtr =
4     std::shared_ptr<ReinforceSchema>;
5
6 using OnionKey = std::tuple
7     <MatProps, bool, double, double,
8     double, ReinforceSchemaSPtr, double, double>;

```

Figure 3.3: Capacity Volume Key format

First, we present and analyze the current design of the Capacity Volume Manager. Then we present our concurrent design. Lastly we talk about thread management issues.

3.2 Methods

3.2.1 The Linear Design

First we study and present the current design of the Capacity Volume Manager. Based on this study we proceed to make decisions on how we can design a concurrent variant. The main component of the Capacity Volume Manager is a key-value store that stores the already calculated Capacity Volumes. We refer to this key-value store as **OnionMap**, since onion is slang for Capacity Volume among structural engineers, due to the onion-like graphical representation of a Capacity Volume. We will also refer to a Capacity Volume as **Onion** and to the Capacity Volume Manager as **OnionManager**. **OnionMap** is implemented as a Hash Table using the *std::unordered_map* STL container.

An element of **OnionMap** is composed of a key and a value. The key is a unique permutation of the values of 8 structural attributes while the value is a shared pointer to a Capacity Volume instance as shown in figure 3.3. We refer to a Capacity Volume key as **OnionKey**.

Throughout C++ code *std::shared_ptr* STL semantics are used to denote **Onion** instances or other resources. *std::shared_ptr* is a smart pointer that retains shared ownership of an object through a pointer. Several *std::shared_ptr* objects may own the same object. The object is destroyed and its memory deallocated when the last remaining *std::shared_ptr* owning the object is destroyed [28]. Usage of *std::shared_ptr* is prevalent in the RAF application code base since it simplifies the memory management of shared resources.

Another important component of the **OnionManager** is the method that allows the user of the Manager to request an **Onion**. We refer to this method as **GetOnion**. The input argument of **GetOnion** is an **OnionKey** while its output is a shared pointer to the calculated **Onion**. We describe the **GetOnion** method in figure 3.4

```
1 class Onion {
2 public:
3     void Solve();
4 };
5 using OnionSPtr = std::shared_ptr<Onion>;
6
7 class OnionManager {
8 public:
9     using OnionMap =
10         std::unordered_map<OnionKey, OnionSPtr>;
11 private:
12     OnionMap onionMap{};
13 public:
14     OnionSPtr GetOnion(const OnionKey& key);
15 };
16
17 OnionSPtr OnionManager::GetOnion(const OnionKey& key) {
18     auto it = onionMap.find(key);
19     if (it != std::cend(onionMap)) return it->second;
20
21     OnionSPtr onion = std::make_shared<Onion>();
22     onion->Solve(key);
23     onionMap.emplace(key, onion);
24     return onion;
25 }
```

Figure 3.4: The `OnionManager` class

from line 17. In lines 18 - 19 `GetOnion` accesses the `OnionMap` and returns the requested `Onion`. If the `Onion` is not present in `OnionMap` then it is not calculated. If it is not calculated, then, in lines 21 - 24, the `OnionManager` proceeds to instantiate it, calculate it by calling its `Solve` method and store it to the `OnionMap`, before finally returning a shared pointer to it.

The calculation of an `Onion` is a time-consuming operation. Each `Onion` instance contains a private method called `Solve`. This method calculates the `Onion` by populating its attribute fields with structural analysis results. We focus our attention on producing a concurrent design that allows for multiple `Onion` instances to run their respective `Solve` method at the same time. The main requirement for this property is data independence among `Onion` instances. After examining the contents of the `Solve` method and its access to any global state, as well as consulting the architects of RAF, we make sure of the fact that multiple concurrent instances of `Solve` are data independent and will not produce data races.

3.2.2 Concurrent Design Requirements

To enable true concurrency we present an `OnionMap` design that allows multiple instances of `GetOnion` to run at the same time utilizing different threads of execution. To achieve this we redesign the execution flow of `GetOnion` while accounting for the following facts

- Concurrent read and write access to `OnionMap` must be supported in order to retrieve or store `Onion` instances.
- The `Solve` method of multiple `Onion` instances, with different keys, must be allowed to run at the same time.
- Thread management must be abstracted from the rest of the application with our design being non-intrusive to the other components of the RAF application.

To guarantee mutual exclusion of code segments we use the concept of locks, implemented as mutex objects using the STL `std::mutex` implementation. `std::mutex` is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. `std::mutex` offers exclusive, non-recursive ownership semantics [22]. To grab or release a mutex we use the `std::unique_lock`. `std::unique_lock` is a general-purpose mutex ownership wrapper allowing deferred locking, time-constrained attempts at locking, recursive locking, transfer of lock ownership, and use with condition variables [31]. Since we make use of condition variables. `std::unique_lock` is an ideal pick for us. With `std::unique_lock` we can follow RAII semantics, where the acquired mutex is released, when the `std::unique_lock` instance goes out of scope.

`OnionMap` is implemented using the `std::unordered_map` STL container. STL containers do not support concurrent write operations by themselves, so we need to

provide mutual exclusion of accessing the container ourselves, while maintaining a valid view of its contents among threads. We use a global lock to provide mutual exclusion when a search/read or modification/write of `UnionMap` takes place. We refer to this global lock as `UnionMapMutex`. This way we eliminate data races on concurrent writes and on cases where a thread reads while another thread is writing.

To handle the parallel calculation of `Union` instances we utilize an additional key-value store, implemented as `std::unordered_map`. We refer to it as `MutexMap`. We use `UnionKey` instances as keys and mutexes as values. This way each stored `Union` in `UnionMap` is associated with its unique mutex in `MutexMap`. Elements are inserted to the `MutexMap` on demand, when a requested key is not already present. With this mechanism we enable the concurrent calculation of `Union` instances. A global lock is used to access `MutexMap`. We refer to it as `MutexMapMutex`. After a mutex acquired, we proceed to access `UnionMap` and retrieve the requested `Union`. If the `Union` is not present in `UnionMap`, we run its `Solve` method and store it before retrieving it.

In the case where, multiple `Union` instances with different keys are requested, their calculation will run concurrently, while `UnionMapMutex` and `MutexMapMutex` serialize the access `UnionMap` and `MutexMap` to respectively, providing synchronization.

In the case where, the same `Union` with a specific key is requested multiple times, all threads except one will block while trying to acquire the `MutexMap` lock associated with the input `UnionKey`. The chosen thread will calculate and store the `Union` instance, then release `MutexMapMutex`. The remaining threads will acquire `MutexMapMutex`, access `UnionMap` and return the already calculated `Union` in a linear fashion.

Furthermore, we need to abstract our design to make it compatible with the rest of the RAF application. Since the application expects the `UnionManager` to complete its requests in a linear fashion, there is no notion of thread joining present. Instead of introducing threading in an external context, we adopt the notion of requestable tasks using `std::promise` and `std::future` STL semantics. `std::promise` provides a facility to store a value that is later acquired asynchronously via a `std::future` object created by the `std::promise` object [27]. `std::future` provides a mechanism to access the result of asynchronous operations [9].

We encapsulate each `Union` to a future and allow the external user to decide when to asynchronously acquire the value of the future, which is the actual `Union` instance. This way we minimize the code changes needed at the external context of our work.

Finally regarding internal thread management we offer two solutions and we give the user the option to pick one at run-time.

- We associate each `Union` request task with a thread execution using the `std::async` facilities of the STL.

- We implement our own thread pool limiting the number of active threads to the user machine CPU cores. Each `Onion` request task is then queued to the thread pool for execution.

The function template `std::async` runs a function asynchronously and returns a `std::future` that will eventually hold the result of that function call [2].

Using the `std::async` facilities simplifies our thread management and complements the usage of `std::future` objects. This way we introduce the notion of asynchronous tasks to our work, without worrying about thread management connotations. The disadvantage of this solution is the Operating System task scheduler. In the event of having multiple time intensive tasks running asynchronously, a lot of time is lost on context switching among different threads on the same CPU core.

By implementing our own thread pool and limiting the number of threads to the number of CPU cores, we eliminate context switching. However, this way we inject an extra level of indirection to our design which results in more room for bugs of concurrent nature. With this solution, we also need to account for the creation and use of promises, since the use of futures is already bound to our design.

3.2.3 The Concurrent Design Implementation

To support our concurrent design, we extend the signature of the `OnionManager` as shown in figure 3.5. In lines 17 - 26, we introduce `MutexMap`, to store our mutexes associated with `OnionKey` instances, as well as the container access mutexes `OnionMapMutex`, `MutexMapMutex`. These mutexes are used to provide mutual exclusion access to `OnionMap` and `MutexMap` respectively. Furthermore, we introduce the `RequestMode` field to switch among implementations at run-time and the private methods `RequestOnion`, `GetOnionConcurrent`, `GetOnionSolve` to aid us in our design. In lines 28 - 29, we change the return value signature of the `GetOnion` method and provide a `set` method to allow the user to switch among implementations at run-time. In lines 32 - 34, `GetOnion` now serves as a wrapper method that acts as an entry point to our design.

The `RequestOnion` method is described in figure 3.6. `RequestOnion` is responsible for abstracting the current `Onion` request as a task, depending on the selected implementation, and returning a future object to the requested `Onion`. The user can decide when to access the wrapped `Onion` object by calling the `.get()` method of the returned future.

- The `SERIAL_LAZY` implementation in line 9 does not spawn any new threads of execution. Lazy evaluation of the task is performed when the user calls the `.get()` method of the returned future. The `Onion` instances are calculated in a serial fashion without introducing any concurrency semantics, hence we call the `GetOnionSolve` method right away.

```

1 using OnionFuture = std::future<OnionSPtr>;
2 using Mutex = std::mutex;
3 using MutexUPtr = std::unique_ptr<Mutex>;
4
5 class OnionManager {
6 public:
7     using OnionMap =
8         std::unordered_map<OnionKey, OnionSPtr>;
9     using MutexMap =
10         std::unordered_map<OnionKey, MutexUPtr>
11     enum class RequestMode {
12         SERIAL_LAZY,
13         CONCURRENT,
14         CONCURRENT_POOL
15     };
16 private:
17     OnionMap onionMap{};
18     Mutex onionMapMutex{};
19     MutexMap mutexMap{};
20     Mutex mutexMapMutex{};
21     RequestMode requestMode{CONCURRENT_POOL};
22
23     auto RequestOnion(const OnionKey& key) -> OnionFuture;
24     auto GetOnionConcurrent(const OnionKey& key)
25         -> OnionSPtr;
26     auto GetOnionSolve(const OnionKey& key) -> OnionSPtr;
27 public:
28     OnionFuture GetOnion(const OnionKey& key);
29     void SetRequestMode(const RequestMode& val);
30 };
31
32 OnionFuture OnionManager::GetOnion(const OnionKey& key) {
33     return RequestOnion(key);
34 }
35
36 void OnionManager::SetRequestMode(const RequestMode& val) {
37     requestMode = val;
38 }

```

Figure 3.5: The Concurrent `OnionManager` class signature

```

1 using OnionPromise = std::promise<OnionSPtr>;
2 using OnionPromiseSPtr = std::shared_ptr<OnionPromise>;
3 OnionThreadPool ThreadPool{
4     std::thread::hardware_concurrency();}
5
6 auto OnionManager::RequestOnion(const OnionKey& key)
7     -> OnionFuture {
8     switch (requestOnionMode) {
9     case RequestOnionMode::SERIAL_LAZY: {
10         return std::async(std::launch::deferred, [this]() {
11             return GetOnionSolve(key);});
12     }
13     case RequestOnionMode::CONCURRENT: {
14         return std::async(std::launch::async, [this]() {
15             return GetOnionConcurrent(key);});
16     }
17     case RequestOnionMode::CONCURRENT_POOL: {
18         OnionPromiseSPtr promise{
19             std::make_shared<OnionPromise>()};
20         OnionFuture future = promise->get_future();
21         ThreadPool.AddJob([this, promise]() {
22             const auto& ret = GetOnionConcurrent(key);
23             promise->set_value(ret);});
24         return future;
25     }
26     default: assert(false);
27 }
28 }

```

Figure 3.6: The RequestOnion method

```

1 using Lock = std::unique_lock<Mutex>;
2
3 auto OnionManager::GetOnionConcurrent(const OnionKey& key)
4     -> OnionSPtr {
5     Mutex* keyMutex{nullptr};
6     {
7         Lock _(mutexMapMutex)
8         mutexMap.emplace(key, std::make_unique<Mutex>());
9         keyMutex = mutexMap.at(key).get();
10    }
11    Lock (*keyMutex);
12    return GetOnionSolve(key);
13 }

```

Figure 3.7: The `GetOnionConcurrent` method

- The *CONCURRENT* implementation in line 13 spawns a new thread of execution for each task. Each `Onion` request acts as a separate thread of execution. Handling of these threads is assigned to the Operating System. With this implementation we do not have a way to handle thread oversubscription.
- The *CONCURRENT_POOL* implementation in line 17 queues up a new tasks for our `ThreadPool`. In line 3 we construct a global `ThreadPool` using the maximum amount of hardware threads. `ThreadPool` is described later in figure 3.9. On this case, we need generate the `OnionFuture` ourselves. In lines 18 - 20, we construct the promise associated with the current task and get its future, which we return in line 24. It is important to construct the promise as a shared pointer, since it is a shared resource among the main thread and the worker thread from our `ThreadPool` as indicated by its capture in the lambda in line 21. The worker thread calculates the `Onion`. When it is done, we notify the associated future, in line 23.

The `GetOnionConcurrent` method is described in figure 3.7. `RequestOnion` is responsible for acquiring the right mutex associated with the current `OnionKey`. In lines 7 - 9 we acquire the `MutexMapMutex` and insert a new mutex associated with the current `OnionKey` to `MutexMap`, if it is not already present. Then we access `MutexMap` to get a pointer to this mutex. In line 10, `MutexMapMutex` is released since the scope of the lock ends. In line 11, we acquire the mutex associated with the current key until the function reaches the end of its execution. In line 12, while the mutex is acquired, we call `GetOnionSolve` and return its result.

The `GetOnionSolve` method is described in figure 3.8. `GetOnionSolve` is responsible for calculating and returning the requested `Onion`. It is an extended

```

1 auto OnionManager::GetOnionSolve(const OnionKey& key)
2     -> OnionSPtr {
3     {
4         Lock _ (onionMapMutex);
5         auto it = onionMap.find(key);
6     }
7     if (it != std::cend(onionMap)) return it->second;
8     OnionSPtr onion = std::make_shared<Onion>();
9     onion->Solve(key);
10    {
11        Lock _ (onionMapMutex);
12        onionMap.emplace(key, onion);
13    }
14    return onion;
15 }

```

Figure 3.8: The `GetOnionSolve` method

implementation of the `GetOnion` method of the serial design of the `OnionManager` that enables mutual exclusion access to the `OnionMap`. In lines 4 - 5, we acquire `OnionMapMutex` and access `OnionMap` to check for the existence of the key. In line 7, after `OnionMapMutex` is released, we check if the key is already present in `OnionMap`. If it is present, then the `Onion` is already calculated, so we return it. Otherwise, in lines 8 - 9, we create an `Onion` instance and execute its `Solve` method. In lines 11 - 12, we acquire `OnionMapMutex` and insert the newly calculated `Onion`. Lastly, in line 14, after the `OnionMapMutex` is release, we return the shared pointer to the `Onion` instance.

3.2.4 The Thread Pool Implementation

For our implementation we delegate thread management to the `std::async` facilities of the STL. By using `std::async`, each asynchronous task spawns a new thread of execution. This leads to the case where each `Onion` request is executed on its separate thread. This leads to thread oversubscription when a large number of `Onion` instances is requested. Since the Operating System is responsible for scheduling which thread to run on which CPU core, we are limited in the ways that we can handle the oversubscription issue.

To tackle this, we extend our implementation to use a thread pool of our own design. We refer to this global thread pool as `ThreadPool`. `ThreadPool` is constructed with a number of worker threads and an empty job queue. A job queue is a queue of tasks to be asynchronously executed by the worker threads. Each thread blocks on a mutex until a job is available on the queue. We wrap `Onion` requests in jobs and delegate them to our `ThreadPool`. By using a `ThreadPool` we

```

1 class ThreadPool {
2 public:
3     using Job = std::function<void()>;
4     using Condition = std::condition_variable;
5
6 private:
7     std::vector<std::thread> threads{};
8     std::queue<Job> jobs{};
9     Mutex jobsMutex{};
10    Condition condition{};
11    bool shutdownPool{false};
12
13    void WaitForJob();
14
15 public:
16    ThreadPool(size_t numThreads);
17    ~ThreadPool();
18    void AddJob(const Job &job);
19 };

```

Figure 3.9: The `ThreadPool` class signature

control the number of running threads, essentially bypassing the oversubscription issue.

The `ThreadPool` signature is described in figure 3.9. In lines 7 - 13 we introduce a `threads` container to store our worker threads as well as a job queue. We refer to the job queue as `Jobs`. We create an abstraction of a job instance using `std::function`, a general-purpose polymorphic function wrapper [8]. We also introduce `JobsMutex`, a mutex to provide mutual exclusion access to our jobs container. We use `Condition`, a condition variable to notify our blocked threads for available jobs and a utility shutdown flag to help with the destruction of our `ThreadPool` instance. `WaitForJob` is a private method which implements a loop for threads to block upon while waiting for a job to available in `Jobs`. In lines 16 - 18, we declare our own constructor and destructor in order to manually manage the working threads. We queue up jobs to the `ThreadPool` by using the `AddJob` method.

The `ThreadPool` methods are described in figure 3.10.

The constructor is described in figure 3.10 from line 1. The input is the number of worker threads present in the `ThreadPool`. These threads are constructed and placed in a container. Each thread is executing the `WaitForJob` method upon construction, waiting for an available job.

The destructor is described in figure 3.10 from line 6. In lines 8 - 9, we acquire `JobsMutex` to raise the shutdown flag. In lines 11 - 13, we proceed to wake up any blocked worker threads and wait for them to join.

```

1 OnionThreadPool::OnionThreadPool(size_t numThreads) {
2     for (size_t i = 0; i < numThreads; ++i)
3         threads.emplace_back([this]() { WaitForJob(); });
4 }
5
6 OnionThreadPool::~~OnionThreadPool() {
7     {
8         Lock_(jobsMutex);
9         shutdownPool = true
10    }
11    condition.notify_all();
12    for (auto& t : threads)
13        t.join();
14    threads.clear();
15 }
16
17 void OnionThreadPool::AddJob(const Job& job) {
18     {
19         Lock_(jobsMutex);
20         jobs.emplace(job);
21     }
22     condition.notify_one();
23 }
24
25 void OnionThreadPool::WaitForJob() {
26     while (true) {
27         Job job{};
28         {
29             Lock_(jobsMutex);
30             condition.wait(1, [this]{
31                 return !jobs.empty() || shutdownPool; });
32             if (jobs.empty()) return;
33             job = jobs.front();
34             jobs.pop();
35         }
36         job();
37     }
38 }

```

Figure 3.10: The ThreadPool class methods

`AddJob` is described in figure 3.10 from line 17. In lines 19 - 20, we acquire `JobsMutex` to insert an element to `Jobs`. In line 22, we wake up one of the blocked worker threads to handle this job.

`WaitForJob` is described in figure 3.10 from line 25. `WaitForJob` is essentially an infinite loop where threads continuously acquire and handle available jobs from `Jobs`. In lines 29 - 31 we acquire the `JobsMutex` and block until a job is available in `Jobs` or the shutdown flag is raised. In line 32, if `Jobs` is empty that means that the shutdown flag is raised, so we end the infinite loop execution by returning. Otherwise, in lines 33 - 34, we pop a job from `Jobs`. In line 36, after we release `JobsMutex` we execute our assigned job.

3.3 Evaluation

We evaluate our work in terms of regression testing and performance profiling. To evaluate our work we create a Testing Pipeline and inject the execution our tests at the startup phase of a development version of the RAF application. This way, we run our regression tests and performance profiling before any major modules of the RAF application are initialized, thus accelerating our work flow. Our regression tests take the form of comparing the contents of calculated `Onion` instances. We compare `Onion` instances calculated by the Concurrent `OnionManager` with their equivalent `Onion` instances, in terms of `OnionKey`, calculated by the Serial `OnionManager`. Our performance profiling takes the form of generating and calculating a large number of keys using the Concurrent and the the Serial `OnionManager` and timing the total execution of the calculations.

Both regression testing and performance profiling is conducted with the help of our own Testing Pipeline.

3.3.1 The Testing Pipeline

Our Testing Pipeline is comprised of `GenerateKeys`, an `OnionKey` instance generation function. `RunBenchmark` is a function that handles the execution of our selected benchmark. Lastly, `ScopeTimer` is a utility class is responsible for inserting time points at selected code sections.

With `GenerateKeys` we are able to generate a collection of `OnionKey` instances by adjusting the value range of each value that is part of the signature of an `OnionKey` instance. We adjust the limits of the value range of each part of an `OnionKey` instance and proceed to generate all the possible permutations based on those limits. The result is a collection of unique keys. Each one of those keys will prompt an `Onion` instance calculation, when supplied to the `OnionManager`. We confirm the structural validity of the values of our limits after consulting the architects of RAF. With these limits we can generate all the possible permutations of unique and valid `OnionKey` instances.

`GenerateKeys` is described in figure 3.11. In lines 1 - 7 and 11 - 24, we configure the values of our `OnionKey` limits, thus tuning the size of our input collection. In

```

1 using SchemaVec = std::vector<ReinforceSchemaSptr>;
2 SchemaVec GetSolvableSchemas {...} // Valid Schemas
3
4 struct Limit {double b, e, s;}; // begin, end, step
5 Limit lim0 {..., ..., ...};
6 Limit lim1 {..., ..., ...};
7 Limit lim2 {..., ..., ...};
8
9 using OnionKeyVec = std::vector<OnionKey>;
10 OnionKeyVec GenerateKeys() {
11     std::vector<MatProps> matV{ ..., ... };
12     std::vector<bool> bV{ false, true };
13     std::vector<double> v0V{};
14     for (double i = lim0.b; i <= lim0.e; i += lim0.s)
15         v0V.emplace_back(i);
16     std::vector<double> v1V{};
17     for (double i = lim1.b; i <= lim1.e; i += lim1.s)
18         v1V.emplace_back(i);
19     std::vector<double> v2V{};
20     for (double i = lim2.b; i <= lim2.e; i += lim2.s)
21         v2V.emplace_back(i);
22     SchemaVec schemaV{ GetSolvableSchemas() };
23     double d0 = ...;
24     double d1 = ...;
25     OnionKeyVec keys{};
26     for (const auto& mat : matV)
27         for (const auto& b : bV)
28             for (const auto& v0 : v0V)
29                 for (const auto& v1 : v1V)
30                     for (const auto& v2 : v2V)
31                         for (const auto& schema : schemaV)
32                             keys.emplace_back(
33                                 { mat, b, v0, v1,
34                                   v2, schema, d0, d1 });
35     return keys;
36 }

```

Figure 3.11: The `GenerateKeys` function

```

1 using OnionVec = std::vector<Onion>;
2 using OnionFutureVec = std::vector<OnionFuture>;
3
4 void OutputResults(const OnionVec& results) {...}
5 //OutputResults serializes onions to file
6 RequestMode requestMode = ...;
7 //SERIAL_LAZY, CONCURRENT, CONCURRENT_POOL
8
9 OnionVec RunBenchmark(const KeyVec& keys) {
10     OnionManager manager{}; //Serial, Concurrent
11     manager->SetRequestMode(requestMode);
12     OnionFutureVec futures{};
13     OnionVec results{};
14
15     {
16         ScopeTimer _("Total Duration");
17         for (const auto& key : keys)
18             futures.emplace_back(manager->GetOnion(key));
19         for (auto& f : futures)
20             results.emplace_back(f.get());
21     }
22
23     return results;
24 }
25
26 const auto& keys = GenerateKeys();
27 const auto& results = RunBenchmark(keys);
28 OutputResults(results); // For regression testing

```

Figure 3.12: The RunBenchmark function

lines 25 - 34, we generate all `OnionKey` instance permutations based on our limits and insert them into a container. Lastly, in line 35, we return our collection of unique `OnionKey` instances.

With `RunBenchmark`, we select an implementation of the `OnionManager` and benchmark it by requesting all the keys present in the input collection and waiting for the calculation of all `Onion` instances.

`RunBenchmark` is described in figure 3.12. In line 4 we define a function responsible for outputting the collection of calculated `Onion` instances to a file. We use this function to perform regression testing to our work by comparing the files containing `Onion` instance results generated by the Serial `OnionManager` to those generated by our Concurrent `OnionManager`. In line 6 we select the implementation of our Concurrent `OnionManager` that we wish to benchmark. In lines 10 - 13

```

1 class ScopeTimer {
2 public:
3     using Clock = std::chrono::steady_clock;
4     using Timepoint = std::chrono::time_point<Clock>;
5
6 private:
7     std::string id;
8     Timepoint begin;
9     Timepoint end;
10 public:
11     ScopeTimer(const std::string& id);
12     ~ScopeTimer();
13 };
14
15 ScopeTimer::ScopeTimer(const std::string& _id) : id{_id} {
16     begin = Clock::now();
17 }
18
19 ScopeTimer::~~ScopeTimer() {
20     end = Clock::now();
21     uint64_t duration = end - begin;
22     //Output to file using id
23 }

```

Figure 3.13: The `ScopeTimer` class

we instantiate an `UnionManager`, apply a selected implementation and construct our `UnionFuture` and `Union` result containers. In line 16 we take advantage of the RAII semantics of our `ScopeTimer` class, described in figure 3.13, to insert timepoints at beginning and end of the local scope defined by lines 17 and 21. Lines 17 - 20 are the actual benchmarked lines where the batch request and calculation of the input `UnionKey` collection is taking place. Lastly, in line 23, we return a collection of the `Union` instance results, in case we want to output them to a file.

In lines 26 - 28 we give the calling convention of our Testing Pipeline.

With `ScopeTimer` we are able to insert timepoints at the beginning and end of a local scope by taking advantage of its RAII semantics.

The `ScopeTimer` class is described in figure 3.13. In lines 7 - 9 we define its id as well as its contained timepoints. In line 16 of the constructor body, we retrieve the timepoints of the class instance construction, from the Operating System. In line 20 of the destructor body we retrieve the timepoint of the class instance destruction, while on line 21 we calculate the lifetime duration of the instance and proceed to output it to a file.

3.3.2 Regression Testing

We need to make sure that the integrity of a calculated `Onion` instances is not invalidated. We conduct regression testing using our Testing Pipeline. To conduct regression testing we first serialize a collection of calculated `Onion` instances to an output text file. We compare the text file produced by calculated `Onion` instances requested by the `Serial OnionManager`, with the text file of `Onion` instances produced by the `Concurrent OnionManager`. These text files contain `Onion` instances that correspond to the same `OnionKey` instances. If the text files are identical, then the `Concurrent OnionManager` produces the same results as the `Serial OnionManager` and consequently, regression testing is a success. We perform regression testing to all the implementations of the `Concurrent OnionManager`. Regarding the size of the generated keys input collection, we make sure we produce every practically possible permutation of `OnionKey` instances, achieving complete input coverage.

3.3.3 Performance Evaluation

To observe the performance benefits of our `Concurrent` design of the `OnionManager` we need to profile it in terms of execution time and *speedup*. We conduct performance profiling using our Testing Pipeline.

Our input collection of unique `OnionKey` instances has the size of 3224, 1612, 806 unique element requests. We profile the time it takes for the `OnionManager` to process the `OnionKey` requests of the above input collection on the following cases

- Using the old `Serial` implementation (aka *OLD*) of the `OnionManager`.
- Using the *SERIAL_LAZY* implementation of the `Concurrent OnionManager`. This way we execute our concurrent methods using only a single thread.
- Using the *CONCURRENT* (aka *ASYNC*) implementation of the `Concurrent OnionManager`. This way each `Onion` request is executed in a separate thread. In this case, we end up with 3224, 1612, 806 threads running at the same time.
- Using the *CONCURRENT_POOL* (aka *THREAD_POOL*) of the `Concurrent OnionManager`, limiting the number of worker threads to 1, 2, 4, 6, 8 on an end-user machine. This way, the number of threads running at the same time does not exceed the amount of the logical cores of our machine.
- Using the *CONCURRENT_POOL* of the `Concurrent OnionManager`, limiting the number of worker threads to 12, 16, 24, 32 on an end-user machine. This way we observe how thread oversubscription affects the performance of our implementation.

While presenting our performance results we use the concept of *speedup* as introduced in the book *An Introduction to Parallel Programming*, on page 58 [24].

$$speedup = \frac{T_{serial}}{T_{parallel}}$$

where T_{serial} is the execution time of the Serial Capacity Volume Manager on a specific number of Capacity Volume requests and $T_{parallel}$ is the execution time of an implementation of the Concurrent Capacity Volume Manager on the same number of Capacity Volume requests.

We also use the concept of *efficiency* as introduced on the same page of the same book.

$$efficiency = \frac{speedup}{p}$$

where p is the number of running threads during the execution time of an implementation of the Concurrent Capacity Volume Manager on a number of Capacity Volume requests. We calculate our *efficiency* on a software thread level, not based on the number of hardware CPU cores of our machine. This way, we can observe the behavior of our *efficiency* on cases of thread oversubscription.

We conduct our performance profiling on two different machines. One of them is a Virtual Machine set up on a NUMA architecture, while the other one is a typical end-user PC.

3.3.4 Evaluation on NUMA architecture Virtual Machine

The specifications of our machine are the following.

- CPU, Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz. Contains 2 NUMA nodes of 14 Physical cores each. Each of the 28 physical cores supports hyper-threading technology totalling to 56 logical cores.
- RAM, 16 GB at 1500MHz
- OS, Microsoft Windows 10 Pro 64-bit

We repeat each measurement 3 times and present an average value.

In figure 3.14 we present the average *speedup* of each of our implementations on a different number of requests. The horizontal axis shows our **OnionManager** implementations while the vertical axis shows the *speedup* when compared with the *OLD OnionManager* design. For the *THREAD_POOL* implementation, we pick the number of 14 worker threads that matches the physical cores of a NUMA node. Each bar color represents a different number of concurrent requests. The red error bars represent the standard deviation of each measurement. The *SERIAL_LAZY* implementation does not offer any *speedup* since the code is executed in a single thread. The *ASYNC* implementation offers an approximate *speedup* of 3.0. Since

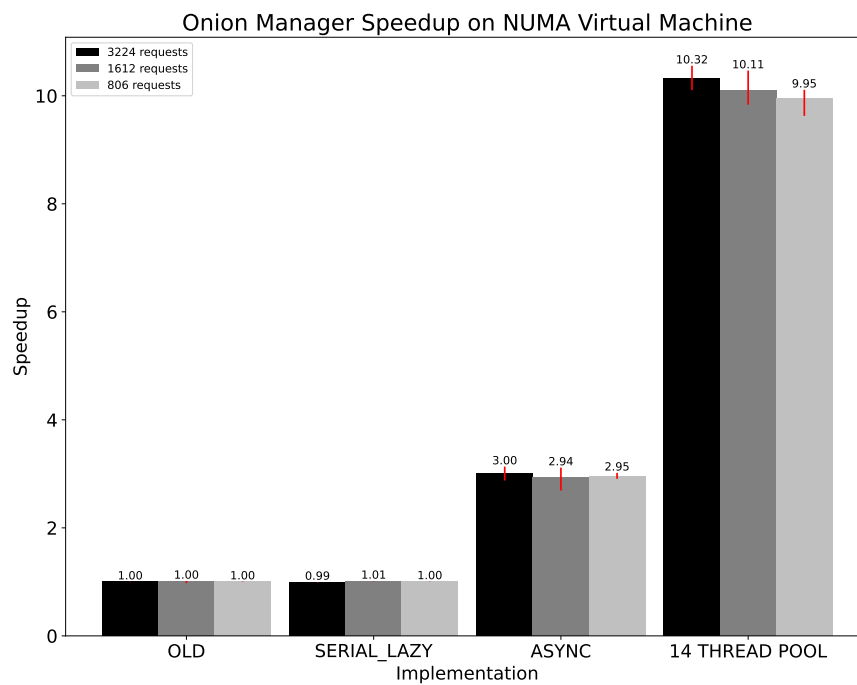
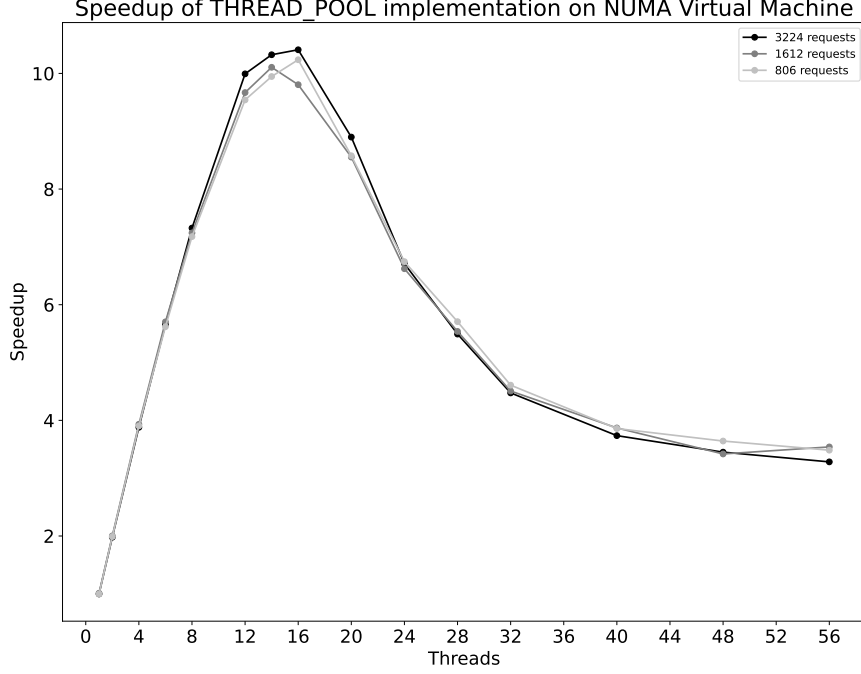


Figure 3.14: *Speedup* of Concurrent Capacity Volume Manager implementations

Figure 3.15: *Speedup* of CONCURRENT_POOL implementation

each request is simultaneously running on a different thread, a large amount of time is spent on context switching and on NUMA node to node communication. The *THREAD_POOL* of 14 worker threads bypasses this issue by achieving a max approximate *speedup* of 10.0. Each thread is mapped on a physical core of a specific NUMA node, eliminating delays due to hyper threading or node communication.

In figure 3.15 we present the average *speedup* of our *THREAD_POOL* implementation on a different number of requests. The horizontal axis shows the number of active worker threads while the vertical axis shows the *speedup* when compared with the *OLD OnionManager* design. Each line color represents a different number of concurrent requests. We do not show the standard deviation of each measurement on this plot. *speedup* increases with the number of threads until it reaches its peak in the approximate value of 10 matching to a thread number of 14-16 threads. Since 14 threads is the number of physical cores for a NUMA node, we notice a decrease in *speedup* from then on due to hyper-threading. After 28 threads, node to node communication comes into play since the number of logical cores per node is exhausted.

In figure 3.16 we present the average *efficiency* of our *THREAD_POOL* implementation on a different number of requests. The horizontal axis shows the number of active worker threads while the vertical axis shows the *efficiency* of the

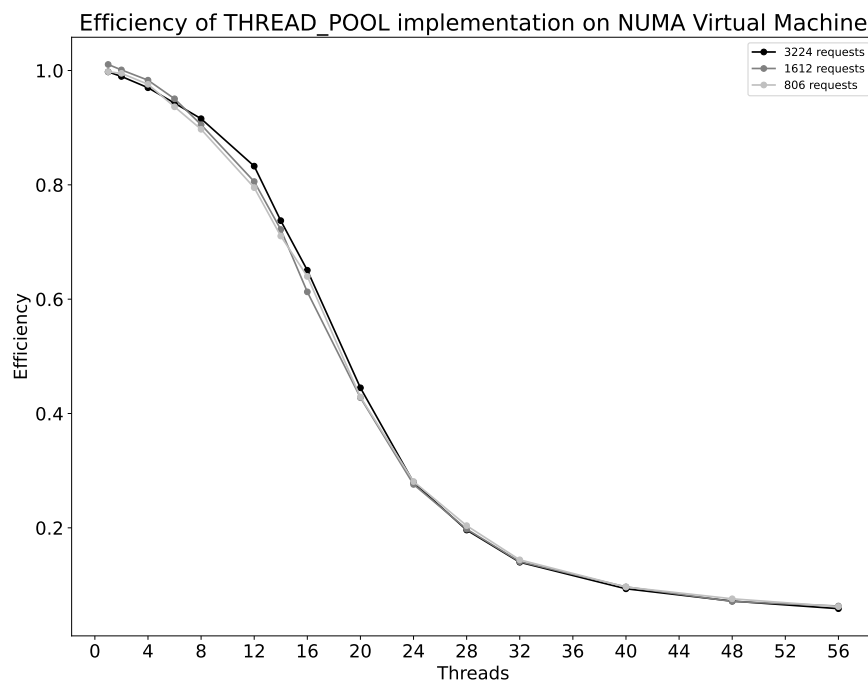


Figure 3.16: *Efficiency* of THREAD_POOL implementation

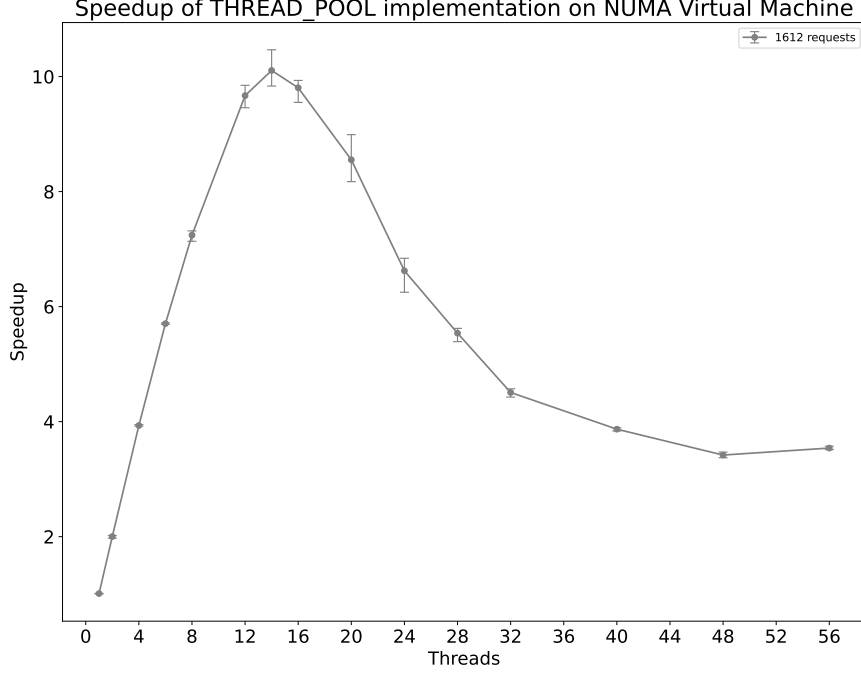


Figure 3.17: *Speedup* of THREAD_POOL implementation on 1612 requests with error bars

corresponding *speedup* value when compared with the *OLD OnionManager* design. Each line color represents a different number of concurrent requests. We do not show the standard deviation of each measurement on this plot. We notice a larger drop in *efficiency* from when exceeding the 14 physical core limit until reaching the 28 logical core capacity per node. This shows that hyper-threading has a larger impact on our implementation than node to node communication.

In figure 3.17 we present the average *speedup* of our *THREAD_POOL* implementation on a 1612 requests. The horizontal axis shows the number of active worker threads while the vertical axis shows the *speedup* when compared with the *OLD OnionManager* design. We present this plot to show the standard deviation of each of our measurements. Our measurements showed large numbers of deviation while hyper-threading on a single node was in effect.

3.3.5 Evaluation on the user PC machine

The specifications of our machine are the following.

- CPU, Intel Core i7-4790S CPU at 3.20 GHz, 4 cores, 8 Logical Processors due to hyper-threading technology

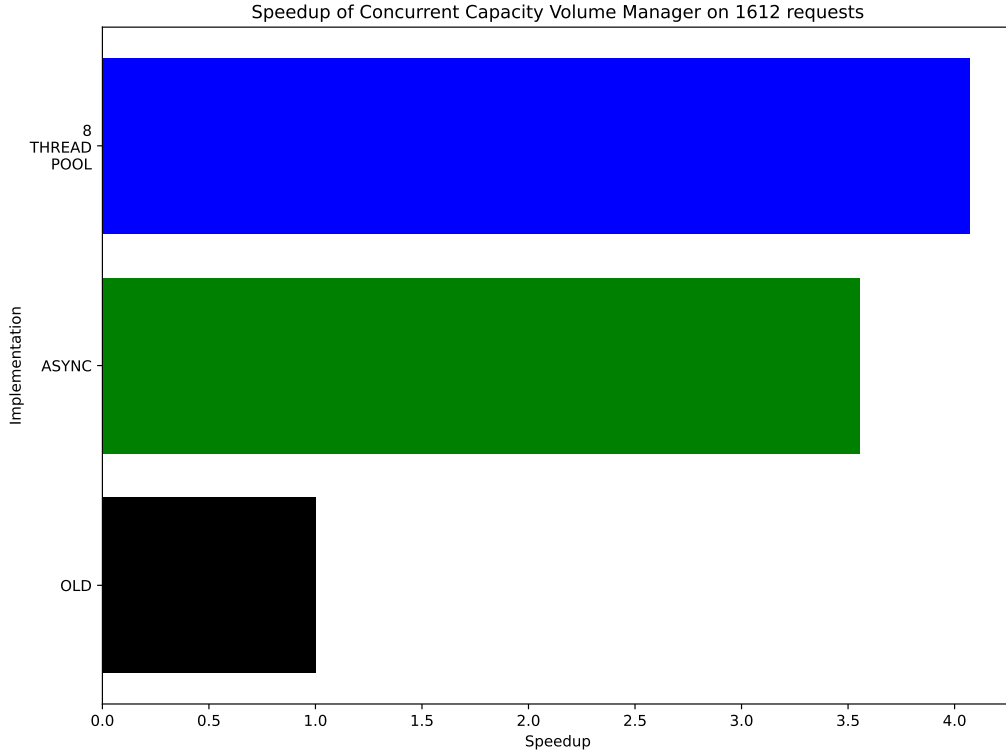


Figure 3.18: *Speedup* of Concurrent Capacity Volume Manager implementations

- RAM, 16 GB at 1500MHz
- SSD, Samsung SSD 860 EVO 250GB
- OS, Microsoft Windows 10 Home 64-bit

Due to time constraints we do not repeat any measurements on our evaluation on the user PC machine.

In figure 3.18 we present the *speedup* of our most efficient implementations on 1612 requests. The vertical axis shows our **UnionManager** implementations while the horizontal axis shows the *speedup* when compared with the *OLD UnionManager* design. For the *THREAD_POOL* implementation, we pick the number of 8 worker threads that matches the logical cores of our machine. The *ASYNC* implementation offers an approximate *speedup* of 3.5 while the *THREAD_POOL* increases it to 4.0. The difference in *speedup* is attributed to context-switching costs on the *ASYNC* implementation.

In figure 3.19 we present the *speedup* of our most efficient implementations on 1612 requests. The vertical axis shows our **UnionManager** implementations while the horizontal axis shows the execution time in seconds. For the *THREAD_POOL*

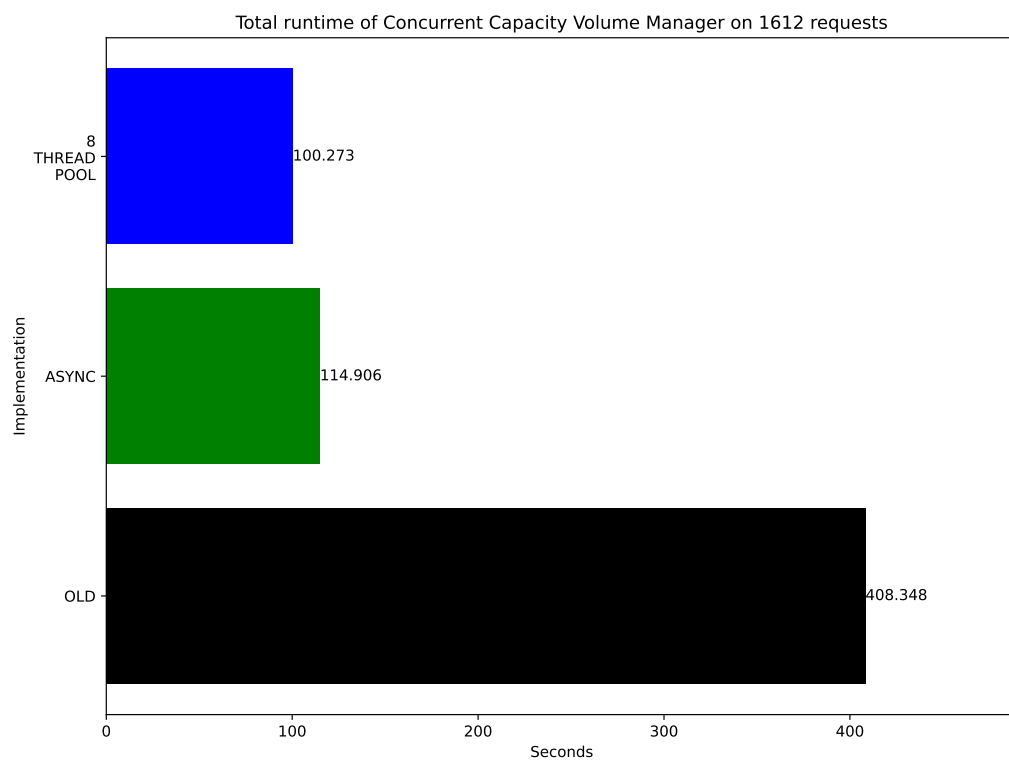


Figure 3.19: Execution time of Concurrent Capacity Volume Manager implementations

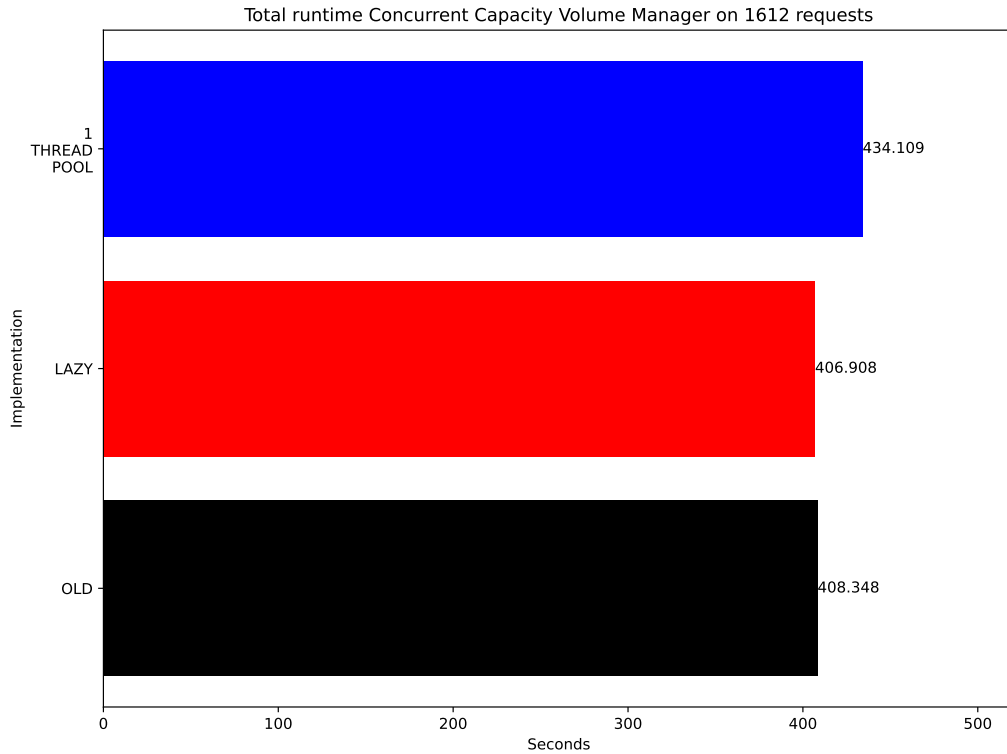


Figure 3.20: Comparison of time lost due synchronization while a single thread is running

implementation, we pick the number of 8 worker threads that matches the logical cores of our machine. The results are analogous to the plot in figure 3.18. Approximately 115 seconds of the *ASYNC* implementation are spend on context switching. This translates to about 13% of the total execution time being spend on context switching.

In figure 3.20 we present the execution time of our single threaded implementations 1612 requests. The vertical axis shows our **OnionManager** implementations while the horizontal axis shows the execution time in seconds. For the *THREAD_POOL* implementation, we pick the number of 1 worker threads, essentially enabling single threaded execution. We notice that about 26 seconds are lost on our worker thread which translates to about 6% of the total execution time spent on synchronization with the main thread.

In figure 3.21 we present the *speedup* of our *THREAD_POOL* implementation on a different number of requests. The horizontal axis shows the number of active worker threads while the vertical axis shows the *speedup* when compared with the *OLD OnionManager* design. Each line color represents a different number of concurrent requests. We do not show the standard deviation of each measurement on this plot. *Speedup* increases with the number of threads until it reaches it's

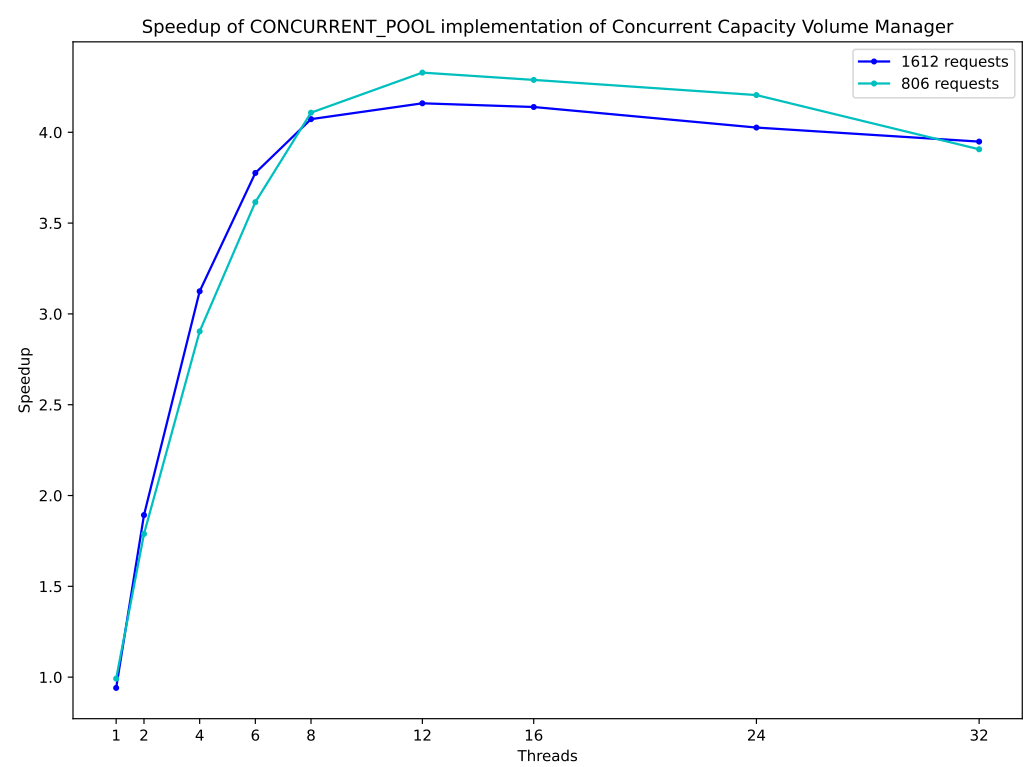


Figure 3.21: *Speedup* of THREAD_POOL implementation

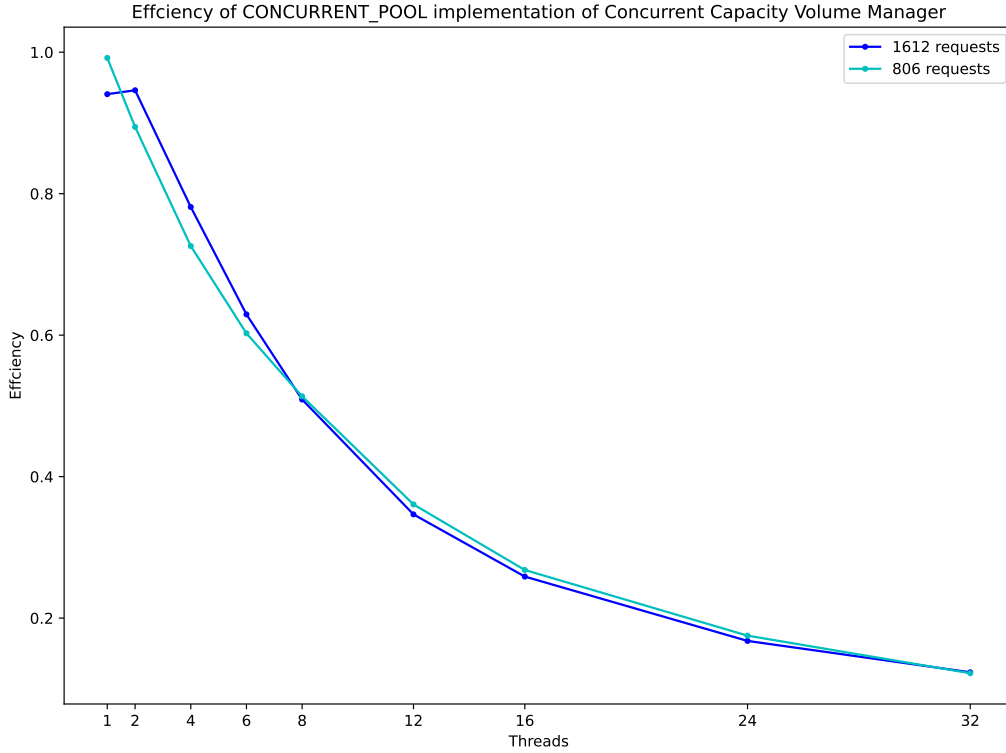


Figure 3.22: *Efficiency* of *THREAD_POOL* implementation

peak in the approximate value of 4.0 matching to a thread number of 8 threads. On this machine we notice an increase in *speedup* even after exceeding the number of our 4 physical cores, utilizing 8 logical cores due to hyper-threading. After 8 cores we do not notice any further increase since oversubscription occurs.

In figure 3.22 we present the *efficiency* of our *THREAD_POOL* implementation on a different number of requests. The horizontal axis shows the number of active worker threads while the vertical axis shows the *efficiency* of the corresponding *speedup* value when compared with the *OLD UnionManager* design. Each line color represents a different number of concurrent requests. The results are analogous to the plot in figure 3.21.

3.4 Related Work

In the book *Patterns for Parallel Programming* [17], *Timothy Mattson, Beverly Sanders, Berna Massingill* describe terminology and thinking methods to applying concurrency in serial programs. A lot of the concepts introduced in this book have been helpful in designing the Concurrent Capacity Volume Manager. The book presents four design spaces in regards to designing concurrent systems. Regarding *Data Decomposition*, we make sure that each Capacity Volume calculation

is abstracted as a data-independent task. While our thread pool is designed in a way that complements Task-Parallelism, we end up achieving Data-Parallelism due to queuing, to our thread pool, the same task of Capacity Volume calculation applied on different Capacity Volume data. *Implementation Mechanisms* such as the synergy of the STL facilities of `std::async` and `std::future` aid us in designing the abstraction of a Capacity Volume request as a task. *Supporting Structures* such as mutexes and thread pools help us achieve mutual exclusion of critical code sections as well aiding us in organising *Task Parallelism*.

Our implementation of our Thread Pool is partly based on *Implementation and Usage of a Thread Pool based on POSIX Threads* by Ronald Kriemann [14]. While we implement the concept of *list scheduling*, we do allow the user to balance the load of our pool.

3.5 Future Work

In our work we present a Concurrent Design of the Capacity Volume Manager that makes it possible to calculate multiple Capacity Volumes at the same time.

We pinpoint a flaw in our *CONCURRENT_POOL* implementation that may reduce the performance gains in certain cases. We present a case where multiple requests for the same Capacity Volume key are queued up one after another. The number of these requests exceed the number of logical cores of our machine. After these requests, other requests for different Capacity Volume keys are queued up. Our thread pool implementation processes job requests in first-come first-served manner. Since multiple requests for the same Capacity Volume key are near the front of the queue, these are the ones processed first. The thread pool does not process any subsequent requests until the first ones, whose size match the number of the worker threads, are finished. Since all requests assigned to worker threads request the same Capacity Volume, only one of them will proceed with its calculation, while the rest will wait its results. This behavior is an intended side effect of the mutex selection related to a Capacity Volume Key. All worker threads will wait on the shared selected Capacity Volume key mutex, minus one thread which will proceed with the Capacity Volume calculation. The other threads will proceed to retrieve the already calculated Capacity Volume, after the calculation is done, and the mutex is released.

We offer two solutions to this problem. The first one is to prioritize and group together requests that correspond to different keys, in the thread pool queue. However, this may break the job abstraction, by injecting code related to Capacity Volumes, in the thread pool implementation. The second is to design a mechanism, which places a currently executing job at the end of the queue if this job is detected to wait on a mutex for a long amount of time.

Chapter 4

Conclusion

We present the way we profile, analyze and optimize the architecture of two legacy code-rich modules of a structural design and analysis application for civil engineers.

On the first module, the Solver, we conduct performance profiling and pinpoint a problematic iteration pattern that provides a CPU bottleneck to the execution time of the module. After we analyze the pattern and present an internal memory layout of the structural element database of the Solver, we design a Memoization Model that eliminates the bottleneck, by replacing the iteration pattern. We present the requirements, interface and provide two implementations of our extensible Memoization Model. Lastly, we discuss the way we future proof our work by linking it to the pre-existing legacy code-base of the Solver module. Furthermore, performance regression tests and evaluate the performance gains of our work using end-user and tool-generated input files. Our Memoization Model achieves up to a 2.0 *speedup* ratio and has already been scheduled for a future commercial release.

On the second module, the Capacity Volume Manager, we present its current design and pinpoint the requirements to extend its design resulting in the Concurrent calculation of Capacity Volumes. Then we present our Concurrent design of the Capacity Volume Manager and its three different implementations. We also present a thread pool implementation that complements one of our Concurrent Capacity Volume implementations. Lastly, we create our own testing pipeline and inject it at the startup of the main application. Through its usage we perform regression tests and evaluate the performance of our work. Our Concurrent Capacity Volume Manager achieves up to a 4.2 *speedup* ratio and has already been scheduled for a future commercial release.

Bibliography

- [1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 14–25, New York, NY, USA, 2003. Association for Computing Machinery. doi:10.1145/604131.604133.
- [2] async. async. URL: <https://en.cppreference.com/w/cpp/thread/async>.
- [3] Autodesk. Autocad, 2022. Last Accessed: April 2022. URL: <https://www.autodesk.com/products/autocad/overview?term=1-YEAR&tab=subscription>.
- [4] R. S. Bird. Tabulation techniques for recursive programs. *ACM Comput. Surv.*, 12(4):403–417, dec 1980. doi:10.1145/356827.356831.
- [5] Yanqing Chen, Timothy A. Davis, William W. Hager, and Sivasankaran Rajamanickam. Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. *ACM Trans. Math. Softw.*, 35(3), oct 2008. doi:10.1145/1391989.1391995.
- [6] CodeSynthesis. Codesynthesis xsd. Last Accessed: April 2022. URL: <https://www.codesynthesis.com/products/xsd/>.
- [7] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, sep 1960. doi:10.1145/367390.367400.
- [8] function. function. URL: <https://en.cppreference.com/w/cpp/utility/function/function>.
- [9] future. future. URL: <https://en.cppreference.com/w/cpp/thread/future>.
- [10] Technical Software House. Tol, 2022. Last Accessed: April 2022. URL: <https://www.tol.com.gr/index.php>.
- [11] John Hughes. Lazy memo-functions. In *Proc. of a Conference on Functional Programming Languages and Computer Architecture*, page 129–146, Berlin, Heidelberg, 1985. Springer-Verlag.

- [12] Intel®. Intel® fortran compiler, 2022. Last Accessed: April 2022. URL: <https://www.intel.com/content/www/us/en/developer/tools/oneapi/fortran-compiler.html#gs.vq7sp7>.
- [13] Intel®. Standard fortran and c interoperability, 2022. Last Accessed: April 2022. URL: <https://www.intel.com/content/www/us/en/develop/documentation/fortran-compiler-oneapi-dev-guide-and-reference/top/compiler-reference/mixed-language-programming/standard-fortran-and-c-interoperability.html>.
- [14] Ronald Kriemann. Implementation and usage of a thread pool based on posix threads., 2004. URL: <https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.465.845>.
- [15] S.K. Kunnath, A.M. Reinhorn, and J.F. Abel. A computational tool for evaluation of seismic performance of reinforced concrete buildings. *Computers & Structures*, 41(1):157–173, 1991. URL: <https://www.sciencedirect.com/science/article/pii/004579499190165I>, doi: [https://doi.org/10.1016/0045-7949\(91\)90165-I](https://doi.org/10.1016/0045-7949(91)90165-I).
- [16] Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, may 1998. doi:10.1145/291889.291895.
- [17] Tim Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 09 2004.
- [18] DONALD MICHIE. “Memo” Functions and Machine Learning. *Nature*, 218(5136):19–22, April 1968. doi:10.1038/218019a0.
- [19] Microsoft. Analyze cpu usage without debugging in the performance profiler (c#, visual basic, c++, f#). Last Accessed: April 2022. URL: <https://docs.microsoft.com/en-us/visualstudio/profiling/cpu-usage?view=vs-2022>.
- [20] Microsoft. Visual studio, 2022. Last Accessed: April 2022. URL: <https://visualstudio.microsoft.com/vs/>.
- [21] Jack Mostow and Donald Cohen. *Automating program speedup by deciding what to cache*. University of Southern California, Information Sciences Institute, 1985.
- [22] mutex. mutex. URL: <https://en.cppreference.com/w/cpp/thread/mutex>.
- [23] openMP. The openmp api specification for parallel programming, 2022. Last Accessed: April 2022. URL: <https://www.openmp.org/>.

- [24] Peter S. Pacheco. Chapter 2 - parallel hardware and parallel software. In Peter S. Pacheco, editor, *An Introduction to Parallel Programming*, page 58. Morgan Kaufmann, Boston, 2011. URL: <https://www.sciencedirect.com/science/article/pii/B9780123742605000026>, doi: <https://doi.org/10.1016/B978-0-12-374260-5.00002-6>.
- [25] Helmut A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer-Verlag, Berlin, Heidelberg, 1990.
- [26] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.
- [27] promise. promise. URL: <https://en.cppreference.com/w/cpp/thread/promise>.
- [28] sharedptr. sharedptr. URL: https://en.cppreference.com/w/cpp/memory/shared_ptr.
- [29] Bentley Systems. Staad.pro advanced 3d structural analysis and design software, 2022. Last Accessed: April 2022. URL: <https://www.bentley.com/en/products/product-line/structural-analysis-software/staadpro>.
- [30] TOL. Raf, 2022. Last Accessed: April 2022. URL: https://www.tol.com.gr/raf/products_raf_main_unit.php.
- [31] unique_lock. unique_lock. URL: https://en.cppreference.com/w/cpp/thread/unique_lock.
- [32] unique_ptr. unique_ptr. URL: https://en.cppreference.com/w/cpp/memory/unique_ptr.
- [33] unordered_map. std::unordered_map, 2000. Last Accessed: April 2022. URL: https://en.cppreference.com/w/cpp/container/unordered_map.
- [34] unordered_map::emplace. unordered_map::emplace. URL: https://en.cppreference.com/w/cpp/container/unordered_map/emplace.
- [35] unordered_map::find. unordered_map::find. URL: https://en.cppreference.com/w/cpp/container/unordered_map/find.
- [36] H. Werner. Schiefe biegung polygonal umrandeter stahlbeton-querschnitte. *Beton- und Stahlbetonbau*, 69(4):92–97, 1974. URL: <https://structurae.net/en/literature/periodicals/beton-und-stahlbetonbau/7847-69-4>.