The Optimisation of a Financial Modelling Code

Mark Tucker

MSc in High Performance Computing

The University of Edinburgh
Year of Presentation: 2009
“All truth passes through three stages.
   First, it is ridiculed.
   Second, it is violently opposed.
   Third, it is accepted as being self-evident.”

Arthur Schopenhauer
*German philosopher (1788-1860)*
Abstract

In the UK, life assurance companies are required by law to demonstrate solvency on a regular basis, and this involves calculating the values of the assets and liabilities which are held by the company. It is frequently the case that these calculations are used as the basis for cashflow projections which may be used by the company to estimate future profitability.

Such projections are commonly performed using computer programs which are generated by specialist valuation software packages. These packages are generally crafted for ease of use, rather than performance of execution, and there is usually only limited scope for improving the performance of the programs they produce.

The aim of this project is to investigate whether it is possible to improve the rate of processing policies by writing programs which perform the same financial projections outside these valuation systems. To achieve this aim, a piece of code which performs the same task will be optimised.

The first stage is to optimise the serial code using compiler options, and then manual optimisation of dominant routines. Once that stage is complete, the code will be parallelised, and the scalability of that parallel code investigated.

It will be shown that;
1) serial optimisation has a major effect, with the actual benefit being dependent on the platform.
2) the code may, depending on the parallelisation strategy adopted, require significant re-engineering.
3) an OpenMP implementation of a task decomposition has very good scaling, and the overall performance of that implementation is only limited by the number of cores available.
4) the overall speedup achieved using a 16-core SMP is in excess of 400x.
Acknowledgements

I would like thank those people who helped make this report possible;
• Dr Alan Simpson, for his guidance throughout this project.
• Graham Stewart, for being so flexible with the time he allowed me to take off work in order to complete this investigation.
• Heather Naismith & Leonie Wittmann, for their suggestions on using various aspects of Office products.

I must also thank my wife, Nicola, for her support throughout the whole period leading up to this report.
# Contents

1) Introduction ................................................................................................................................................. 1
   1.1) The Program Under Investigation ........................................................................................................ 1
   1.1.1) Reserve Calculations ......................................................................................................................... 1
   1.1.2) Profitability Calculations ................................................................................................................. 1
   1.2) The Valuation System Used ............................................................................................................... 2
   1.2.1) Overview .......................................................................................................................................... 2
   1.2.2) Advantages and Disadvantages of the System Used ....................................................................... 3
   1.3) Initial Code for this Investigation ..................................................................................................... 4
   1.4) Data Used in this Investigation ......................................................................................................... 4
   1.5) Platforms Used in this Investigation ................................................................................................. 5
   1.6) Timing Methodology .......................................................................................................................... 6
   1.7) Testing and Verification ..................................................................................................................... 7
   1.8) Approach to Enhancing Performance ............................................................................................... 8

2) Program Structure, Initial Profiling and Compiler Flags ................................................................................. 8
   2.1) Structure of the Original Code ......................................................................................................... 8
   2.1.1) The Valuation System ..................................................................................................................... 8
   2.1.2) The Code in this Investigation ...................................................................................................... 9
   2.2) Profiling .............................................................................................................................................. 10
   2.2.1) The Initial Profile .......................................................................................................................... 10
   2.2.2) Profiling i/o .................................................................................................................................... 11
   2.3) Compiler Flags .................................................................................................................................... 12
   2.3.1) The Valuation System ................................................................................................................... 12
   2.3.2) The Code in this Investigation ...................................................................................................... 12
   2.3.3) Investigating the Components of /fast ......................................................................................... 13
   2.4) Initial Simplification of the Code ...................................................................................................... 14
   2.4.1) Debugging Mode vs. Live Mode .................................................................................................... 14
   2.4.2) Variable Declarations ................................................................................................................. 15
   2.4.3) Number of Polices Processed ....................................................................................................... 15
   2.4.4) Effect on Performance .................................................................................................................. 15
   2.5) Summary .......................................................................................................................................... 15

3) Manual Optimisation of the Interpolation Routine ...................................................................................... 16
   3.1) Re-profiling the Code ....................................................................................................................... 16
   3.2) Use of the Interpolation Function .................................................................................................... 17
   3.3) Optimising the Function ................................................................................................................... 19
   3.3.1) Using Features of the F90 Language .............................................................................................. 19
   3.3.2) Unrolling the Loops ...................................................................................................................... 19
   3.3.3) Cleaning the Code ........................................................................................................................ 20
   3.3.4) Permutations of Initialisation Statements .................................................................................... 21
   3.3.5) Speed of Light .............................................................................................................................. 21
   3.4) Effect on Performance ...................................................................................................................... 22
   3.5) Investigating the Permutations of Initialisation Statements ................................................................ 23
   3.5.1) Possible Permutations ................................................................................................................. 23
   3.5.2) Future Investigation of the Optimal Permutation ....................................................................... 25
   3.6) Summary .......................................................................................................................................... 25

4) Manual Optimisation of the Reserve Calculation ......................................................................................... 26
   4.1) Re-profiling the Code ....................................................................................................................... 26
   4.2) Use of the Reserve Calculation ....................................................................................................... 27
   4.3) Optimising the Reserve Calculation .................................................................................................. 29
   4.3.1) Using Features of the F90 Language .............................................................................................. 29
   4.3.2) Removing Array Initialisations ................................................................................................... 29
   4.3.3) Shortening Arrays ........................................................................................................................ 30
   4.3.4) Simplifying the Calculations ........................................................................................................ 30
   4.3.5) Calculation of Discount Factors ................................................................................................. 31
   4.3.6) Speed of Light .............................................................................................................................. 32
   4.4) The Effect on Performance .............................................................................................................. 32
   4.5) Replacing the Power Calculations ................................................................................................... 33
   4.5.1) How the Power Calculations were Replaced ............................................................................... 33
   4.5.2) Validation of Number of Power Calculations Replaced ............................................................... 34
List of Figures

Figure 1.1: Schematic representation of a typical valuation package ......................................................... 3
Figure 2.1: Imitation of the ‘Variable Entry’ screen in a valuation package ............................................. 8
Figure 2.2: Initial profile of code under investigation ................................................................................. 11
Figure 3.1: Profile after tuning compiler options ........................................................................................ 17
Figure 3.2: Example of interpolation in \( l_x \) ................................................................................ 18
Figure 4.1: Profile after manual optimisation of interpolation routine ....................................................... 27
Figure 5.1: Profile after manual optimisation of reserving calculation .................................................... 38
Figure 5.2: Progression of profiling information ......................................................................................... 39
Figure 5.3: Cumulative speedup during serial optimisation ...................................................................... 43
Figure 5.4: Speedup resulting from individual phases of serial optimisation .......................................... 44
Figure 6.1: Processing rate for varying numbers of policies and bite sizes ............................................... 52
Figure 7.1: Speedup for 1000 policies on Ness ........................................................................................... 54
Figure 7.2: Scaling with problem size when using 4 cores on the desktop ............................................. 56
Figure 7.3: Scaling with problem size when using 4 cores on Ness ......................................................... 57
Figure 7.4: Scaling with problem size when using 16 cores on Ness ....................................................... 58
Figure 8.1: Cumulative speedup during serial optimisation ................................................................. 61
Figure 8.2: Scaling on the desktop as problem size increases ................................................................. 62
Figure 8.3: Scaling on Ness as problem size increases ........................................................................... 62
Figure 8.4: Progression of processing rate as optimisation progressed .................................................... 63
List Of Tables

Table 1.1: Approximate distribution of escalation rates in the synthetic data ........................................................5
Table 2.1: Time to process 1000 policies with various compiler options ............................................................13
Table 2.2: Combinations for components of /fast ............................................................................................13
Table 2.3: Time to process 1000 policies using various combinations of components of /fast .......................14
Table 3.1: Time to process 1000 policies as optimisation of interpolation routine progressed ......................22
Table 3.2: Time to process 1000 policies for permutations of statements in interpolation routine .................24
Table 3.3: Grouping of timings for permutations in interpolation routine .......................................................24
Table 4.1: Time to process 1000 policies as optimisation of reserving calculation progressed .....................32
Table 5.1: Time to process 1000 polices using various compiler options ............................................................40
Table 5.2: Time to process 1000 polices during manual optimisation of the interpolation routine ...............41
Table 5.3: Time to process 1000 polices during manual optimisation of the reserving calculation .................42
Table 6.1: Time to process 1000 polices during re-engineering ........................................................................48
Table 6.2: Time to process 1000 polices using various numbers of threads and different schedules ...........49
Table 6.3: Time to process 1000 polices using various bite-sizes .......................................................................51
Table 6.4: Time to process varying numbers of policies and bite sizes ...........................................................51
Table 7.1: Time to process 1000 polices, using the parallelised program, on the desktop ...................................53
Table 7.2: Time to process 1000 polices, using the parallelised program, on Ness ..............................................53
Table 7.3: Processing rate, using the dynamic(5) schedule, for 1000 polices .......................................................55
Table 7.4: Processing rate, as the number of polices processed increased, on the desktop .............................56
Table 7.5: Processing rate, as the number of polices processed increased, on Ness .............................................58
Table 8.1: Parameters for parallel runs in Figure 8.4 ......................................................................................63
List Of Extracts

Extract 2.1: Code for ‘cumulative PVRE’ produced by valuation package ............................................................ 9
Extract 2.2: Code for ‘cumulative RE’ produced by valuation package ................................................................. 9
Extract 2.3: Profile of original code ......................................................................................................................11
Extract 2.4: iFort’s basic optimisation flags ........................................................................................................12
Extract 3.1: Profile after tuning compiler optimisations and initial simplification ............................................ 16
Extract 3.2: Original implementation of the interpolation function ................................................................. 18
Extract 3.3: Initialisation of arrays in the interpolation function ............................................................................19
Extract 3.4: Fully unrolled calculation of lagrangePoly(k) .................................................................................19
Extract 3.5: Simplification of unrolled calculation of lagrangePoly(k) .............................................................. 20
Extract 3.6: Further simplification of unrolled calculation of lagrangePoly(k) .................................................... 20
Extract 3.7: Replacement of floating-point indices ..............................................................................................20
Extract 3.8: Initialisation statements within the interpolation routine .............................................................. 21
Extract 3.9: Rearranged initialisation statements .................................................................................................21
Extract 3.10: Interpolation at the speed-of-light ................................................................................................. 21
Extract 3.11: The original version of the optimised interpolation routine .......................................................... 23
Extract 3.12: Labelled statements in the body of the interpolation routine ......................................................... 23
Extract 3.13: Optimal ordering of statements in the interpolation routine ....................................................... 24
Extract 4.1: Profile after manual optimisation of the interpolation routine ..........................................................26
Extract 4.2: The start of the annuity reserve {single life} function ..................................................................... 29
Extract 4.3: Initialisation statements considered for rearrangement ................................................................. 30
Extract 4.4: Calculation of probabilities ........................................................................................................... 31
Extract 4.5: Simplified calculation of probabilities ............................................................................................ 31
Extract 4.6: Final summation of annuity factor ................................................................................................. 31
Extract 4.7: Calculation of discount factors using an array ............................................................................. 34
Extract 4.8: Timing power and multiplication calculations ............................................................................... 35
Extract 5.1: Profile after manual optimisation of reserve calculation ............................................................. 37
Extract 6.1: Pseudocode for implementation of data decomposition ............................................................. 47
Extract 6.2: Pseudocode for processing using multiple bites ............................................................................. 50
1) Introduction

In this Chapter we introduce some of the rationale behind the program investigated in this report; why it is needed, and what it does. We then discuss the system used in a commercial environment, along with some of its advantages and disadvantages. Since this investigation is not performed in that commercial environment, we cannot use the actual program used within the business; that program is of commercial interest, and is subject to intellectual property constraints. We therefore need to consider the code actually investigated, and how it differs from the code used within the business.

We then introduce the data used in this investigation; why it is not actual data from the commercial environment, what the data used here represents, and how it was created. Next, we present the platforms used; what they are, and their purpose in this investigation. Finally, we discuss the timing methodology and the testing used during this investigation.

1.1) The Program Under Investigation

All insurance businesses operating within the UK are regulated to ensure that, among other things, they are solvent. At its simplest, solvency boils down to holding sufficient assets {cash, stocks, bonds, Gilts, etc.} so that the income from those assets {both regular interest, dividends, etc. and proceeds from their disposal}, together with future premium income, are sufficient to meet the liabilities arising from, and the expenses attributable to, the policies to which those assets relate. This can be interpreted simplistically as

\[
\text{solvency} \leftrightarrow \begin{cases} 
\text{current value of (assets and income) exceeds} \\
\text{current value of (liabilities and expenses)}
\end{cases}
\]

Further interest in the values of the assets and liabilities lies in the fact that the timings of the cashflows, and the order in which they happen, can be used to estimate the future profit arising from the policies.

In this investigation we consider annuity policies, which could be considered as the simplest type of life assurance policy. An annuity is simply “a stream of payments” [1]. As an example, consider a bank loan in which the bank ‘give’ the customer an amount of money now, in exchange for regular future repayments; the bank has actually bought an annuity from the customer.

This principle is far more common than just bank loans and mortgages. Consider a pensioner who takes his retirement fund to an assurance company and purchases an annuity; the annuity is actually the regular payments the pensioner receives from the assurer in the form of their pension. This is technically a life-annuity because payments depend on the survival of the policyholder [2]. It is this last situation we consider in this report.

1.1.1) Reserve Calculations

In this investigation we consider only the liabilities of the life assurance office; i.e. the payments to the policyholder and the associated expenses. This is because, in the reality of the commercial environment which inspired this investigation, the assets are valued separately, on a completely different system.

The present value of the future payments to the policyholder can be calculated using methods which are elementary to actuarial science [2]. The basic concept is to allow for interest accruing on the current funds held, and for the probability of making each payment. The actual calculations used in the program under investigation will be discussed further in Section 4.2.

1.1.2) Profitability Calculations

The estimation of profit which can emerge from one of these policies requires knowledge of future reserves. Each year payments will be made to the policyholder, expenses will be paid, and income will be received on the funds held. Crucially, because payments are made throughout the year, the reserve required at the end of the year will, under normal investment conditions, be lower than the reserve required at the start of the year. This
release of reserves dictates that all those reserves derived in Section 4.2 do need to be calculated for each future step of the projection.

Moreover, to increase the accuracy of the information available for running the business, and to allow for the fact that most pensioners now receive their payments more frequently than yearly, the whole projection of all of this business is performed using monthly steps rather than yearly. While this does add to the accuracy, it also adds to the calculation count, and hence to the run-time.

Industry-standard software is used to produce code which runs under Windows XP on PCs with 2.99GHz hyper-threading CPUs. In this environment we see a processing rate of roughly 1 policy per second. Even allowing for the complexity of the calculations, and for the fact that the commercial software used cannot utilise multi-core processing, this processing rate feels slow ... over a day to process a single dataset of 100,000 policies seems somewhat archaic.

1.2) The Valuation System Used

Most computer users understand that one of the main uses for a spreadsheet is to allow numbers to be entered, stored and manipulated so that summaries and reports can be produced according to the manner in which those numbers entered are manipulated. Clear benefits to spreadsheets are their flexibility, and the fact that users do not necessarily need to fully understand the calculations which are performed within functions provided by the spreadsheet developers.

Within the life assurance industry in the UK, several valuation packages are available. The idea behind these financial valuation packages is analogous to the spreadsheet example, and, as such, these valuation packages are, in concept, very good.

1.2.1) Overview

At their most basic level these packages require the relationships between variables to be entered and, using these relationships, the valuation software performs the relevant data-flow analysis and produces source code in a ‘normal’ programming language, which is often C, C++ or Fortran, although Visual Basic has recently appeared as the language used by one valuation package. This code is then compiled and linked into an executable using standard compilers and linkers which are controlled by the valuation package issuing command-line instructions internally. The executable produced by this process is then launched by the valuation package, and when execution is complete the valuation software ensures that any memory used has been correctly freed.

In general, the functionality of a spreadsheet can be enhanced using add-ins, macros and user-functions. Analogously, it is generally possible to create user-functions for these financial valuation packages. While these functions can be built to perform calculations of arbitrary complexity, they have a minor drawback in that they need to be written in one of the standard languages mentioned above. Since these functions lie outside the normal functionality of the package, they need to be written and compiled by a competent programmer before they can be used by the valuation package. They are only integrated into the system at the point at which the package performs its linking stage.
1.2.2) Advantages and Disadvantages of the System Used

Given this mode of operation, the advantages to these systems are clear. For normal operation, they allow non-programmers to produce some really quite complex programs which can be used to estimate the current value of, or profitability within, some quite complicated assurance contracts; the user simply needs to input the correct relationships and the valuation package does the rest. Professional software developers only need to get involved when relationships between variables become too complex to enter into the system, and user-functions need to be built.

An added advantage to storing the relationships between variables, rather than actual source code, is that it becomes reasonably straightforward for non-programmers to change the way policies are processed; the user simply needs to change the relevant relationships between the variables and, assuming that no errors were introduced, the valuation package will produce a new executable. The advantage here is that, for this mode of use, users do not need to become involved in the details of variable types, or the shapes of arrays, etc.

However, while the disadvantages of these systems are not quite so obvious, they are as equally fundamental as the advantages. A slight issue arises from the fact that even if the value of only one parameter is changed in only one place, then the system will generate totally new source code for the relationships entered by the non-programmers, i.e. not the user defined functions, re-compile all of that source, and link the resultant object file. This is clearly far less efficient than being able to simply re-compile only those parts of the source code which have changed.

Another issue is that, for the relationships entered by the non-programmers, there is no possibility of being able to tune compiler, or linker, optimisations. As a consequence of this, all programs depend on compiler settings which were ordained by the writers of the valuation package at the time it was developed. While these standard options may be beneficial in some situations, they are certainly not optimal for all codes.
A far more fundamental problem is that the user is constrained by pre-configured parameters which were included in the package at the time it was built, e.g. some of these valuation packages only allow a forward progression through time. The only way to work around such constraints is to build relationships which are at a far more fundamental level; i.e. move away from the text-based relationships and utilise the skill of the programmers. It is this problem which is the predominant motivation for the investigation in this report.

1.3) Initial Code for this Investigation

We have, up to this point, intentionally tried to keep this introduction as general and anonymous as possible. However, anyone who has a reasonable knowledge of these valuation packages will, from what follows, be able to identify the system which motivated this investigation. For this, we apologise in advance; our intention is not to decry any particular system, but rather to investigate whether there is any benefit in moving away from these packages and writing *ab initio* programs using a language which is suitable for computationally intensive applications.

At the time this investigation was first mooted, the package used within the commercial environment transforms the stored relationships into Fortran77 source code, and then compiles it using Compaq Visual Fortran V6.6 (CVF). This leads to two major issues when producing a piece of code which replicates that produced within the commercial environment. Firstly, following many years of writing modular code, we find it extremely difficult to think in a manner which enables the writing of large swathes of sequential procedural code. Secondly, and more fundamentally, because CVF is no longer available, we cannot obtain a licensed copy to use outside the office. The combination of these two factors means that our starting point is some Fortran90 code which runs on a laptop after having been compiled using Intel Visual Fortran V10.1.

The code used in this investigation is a mirror of the code used in the commercial environment in that it will, starting from the same combination of initial parameters, produce the same results as the system used within the office. However, to ensure that there is no confusion about the code in this investigation, it is worth re-iterating that the code on which this investigation is based is F90 written specifically for this project; it is not the F77 produced in the business environment.

1.4) Data Used in this Investigation

We cannot use real data from the commercial environment for the same reason we cannot use the real code; this investigation is performed outside the business. Therefore, since this investigation is not the purpose for which the data was collected, use of that data is precluded by the Data Protection Act. We have therefore created some data to use in this investigation; having been artificially created, this data will be referred to as ‘synthetic data’.

In order that the performance of the program investigated is representative of the performance of the valuation system, our synthetic data look like they could be drawn from a population of policyholders with similar characteristics as those in the business. Because this investigation considers annuity policies, payable to pensioners, our data was created so that it represents a similar group of pensioners, even if they are fictitious.

This data was created so that all attributes are random, and the main characteristics, which match typical data from the business, may be summarised as follows:

1) the date of birth is uniformly distributed between 1/1/1942 and 31/12/1951. For this investigation, all calculations are performed as at 31/12/2008, and so the age at valuation is uniformly distributed between 57 and 67. Therefore, our data represents a cohort of people who have retired recently, either at normal retirement age, or slightly early.

2) the policy inception date is uniformly distributed between 1/1/2007 and 31/12/2007 so that, when considered against a valuation date of 31/12/2008, these polices represent a cohort of recent business.

3) roughly 73% of the policyholders are male.

4) roughly 81% of the policies have payments made monthly, the remainder having payments made annually.
5) if we define $s$ as the modal payment, i.e. the amount paid at each payment \( \frac{\text{annual amount}}{\text{payments per year}} \), then $s$ has a log-normal distribution with a mean of roughly 5.0 and standard deviation of about 1.48, i.e. \( \ln s \sim N(5.01, 1.477^2) \).

6) the rate of escalation, i.e. the annual increase in the amount paid to the policy holder, is approximately distributed as

<table>
<thead>
<tr>
<th>Escalation Rate</th>
<th>0%</th>
<th>3%</th>
<th>4.25%</th>
<th>5%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proportion of Policies</td>
<td>95.2%</td>
<td>3.5%</td>
<td>0.8%</td>
<td>0.5%</td>
</tr>
</tbody>
</table>

Table 1.1: Approximate distribution of escalation rates in the synthetic data

In order to create our data, we used standard statistical simulation techniques [3]. Using the probability density function for a required distribution, we can find the cumulative distribution function and hence the probability that a variable from that distribution is less than a specified value. Generating a random number in the range from zero to one produces the required probability, which we apply to the cumulative distribution function to provide the relevant value from the distribution.

Clearly, this method relies on the availability of a random number generator; we have used the \texttt{rand()} function provided with Fortran. As we shall see in Section 1.5, the Fortran compiler on our main platform is Intel’s \texttt{iFort}, and their implementation of \texttt{rand()} is a linear congruential generator. As such, the deterministic sequence of numbers produced lacks many of the desirable properties of random numbers. However, the sequence produced by \texttt{rand()} appears to be uncorrelated to sufficient lag that it suffices for our current purpose, i.e. creating a set of data which represents a cohort of pensioners. Clearly, if we were performing a Monte Carlo simulation in a physics problem then we would use a generator which produces sequences which have more desirable properties, e.g. a lagged Fibonacci generator.

1.5) Platforms Used in this Investigation

The valuation package used in the commercial environment is an application which runs under Windows XP, and the fact that the package launches the executables it produces means that these executables are only run in the same Windows environment as the package. Being a commercial environment means that great importance is placed on information security and data protection. A direct result of this is that there is a plethora of anti-virus, data-tracking, and other security utilities running on the system.

Further, given their importance to the business, these packages, the executables they produce and, to ensure the safety of the data, the input data and the results, are stored remotely using an Ethernet network; nothing is stored locally on the PC. Within the office, these packages and the executables they produce are run on PCs using 2.99GHz hyper-threading processors, although, as mentioned in Section 1.1.2, the executable produced by the valuation package is single-threaded.

This combination of single-threadedness, the number of security daemons running, and the need to access data across a network, leads to a rate of processing of approximately 1 policy per second. The business has a total of about 160,000 life-annuity policies of the type considered in this report, and the overall time to process all these policies using the valuation system is roughly 44 hours. For purposes of reporting management information, these policies are partitioned into separate files, the largest of which contains roughly 100,000 policies and takes about 28 hours to process.

Because this investigation is not part of the company’s business, we cannot perform this investigation on their systems, and so the normal environment in which these programs run cannot be accurately reproduced. However, to try to emulate the ‘unclean’ nature of that environment we use a Lenovo 3000 N200 laptop which has a 2.0GHz dual-core processor and a straight-out-of-the-box installation of Windows XP onto which Intel’s Fortran compiler, \texttt{iFort}, has been installed. The untuned nature of the operating system on this machine is an attempt to have the laptop running with as many daemons as the PCs within the business. This platform does in fact produce a similar rate of processing; roughly 1 policy per second for the initial code.

This investigation was conceived in a business environment where Windows is the only available operating system. There is therefore an interest in how well the code could perform on a Windows platform which has been set up so that computationally intensive applications are not hampered by the environment in which they
are running. Therefore EPCC provided a Dell ‘Precision T3400’ desktop PC which runs Windows XP using a 2.6GHz quad-core CPU, and has iFort installed. The interest in this machine comes from discovering how well a parallelised code scales within Windows on PCs similar to those currently on the market.

We are also interested in discovering how well the parallelised code scales to the numbers of cores which could be available in PCs in the near future. To emulate those machines we used Ness, EPCC’s multi-core training machine which can be used with up to 16 cores in shared-memory mode. Ness runs scientific Linux [4], has Portland’s PGF90 compiler rather than Intel’s iFort, and has 2.6GHz dual-core CPUs. The reason for including this platform was to discover how well the code performs on a machine whose processing equivalent could soon be available to the commercial environment, i.e. PCs with multiple quad-core CPUs, which can be run as shared-memory boxes.

In this investigation, we are primarily interested in whether we can improve performance by producing readable, maintainable code, and then tuning that code. Comparisons between platforms are not the main topic of interest; although such comparisons could prove to be an interesting diversion, we are primarily interested in improvements within a particular platform.

1.6) Timing Methodology

We mentioned in Section 1.5 that the initial code processes at a rate of approximately one policy per second. This rate was derived as the result of processing 1000 policies, where the elapsed time was roughly 988 seconds.

It is apparent that, for intervals on this scale, the Fortran system clock, which can generate a precision of the order of milliseconds, is sufficient. We considered the use of timers such as timef() and rtc(), which can generate a precision of the order of microseconds, but concluded that such a level of precision is not required because the proportional increases in performance do not need to be quoted to, say, 8 significant figures; even the four significant figures achievable from milliseconds are verging on being excessive.

The Fortran system_clock() routine has the added advantage of portability; a code which has been developed to use it should run on any platform using F90. This means that our code can be used, without the need to alter the timing mechanism, on any of the platforms introduced in Section 1.5.

We mentioned above that the initial timing obtained was the result of processing 1000 policies. For consistency, we will continue to use the same set of 1000 policies, unless otherwise stated. Clearly, by processing the same set of policies, the timing produced will be comparable because they are considering the same overall task, i.e. process that set of policies. However, to allow for different daemons running at different times, leading to variation in runtimes, each timing obtained is the result of running the program multiple times in order to estimate the variability of the runtimes.

We obtained the ‘spread’ of the repetitions of the timings as

\[
\text{spread} = \frac{\max(\text{runtime})}{\min(\text{runtime})}
\]

Clearly, a spread value close to 1 indicates that there was little variation in the timings, and in these cases, we report the minimum runtime since that represents the best performance which was achieved.

At the outset of this investigation, we executed and timed each version of the program seven times. However, it soon became apparent that the timings were so consistent that four repetitions of the execution were sufficient to indicate whether the runtimes were stable, and reproducible; we therefore used only four runs for the remainder of the investigation.

1.7) Testing and Verification

A naive statement of the aim of an optimisation project is ‘to make the program run faster’. However, it is possible to make any program run arbitrarily fast by simply removing large parts of the code, thereby allowing the program to output rubbish. Therefore, a better statement of the aim of an optimisation project is ‘to make the program run faster while producing the same results’.
The test data described in Section 1.4 were run through the live F77 program on the system in the business and the results from that run have been compared to the output from running that same test data through the initial version of the new F90 code investigated on the laptop. That comparison showed the output for each policy to be the same to 6 significant figures in each of the two environments. Within the commercial environment which inspired this investigation, any difference in results on different platforms is considered to be immaterial if it is less than 1%. Therefore, an actual relative difference of less than $10^{-4}$ provides evidence that the initial program for this investigation does give the same results as the real program.

This regression test will be performed at the end of each stage of the investigation; we will compare the output from the current stage against the initial output to ensure that any changes made during the current stage have not affected the output.

1.8) Approach to Enhancing Performance

Grama et al. [5] state that “Parallelisation of sub-optimal serial programs often has the undesirable effects of unreliable speedups and misleading runtimes”. This investigation is an optimisation project, and so we follow the standard process which is applied to scientific and engineering codes; we perform serial optimisation of the code before we add parallelism.

Initially we profile the code to see whether there are any hotspots, and if so where they are. We then investigate compiler options to see whether we can improve performance without altering the code. Following that, we manually optimise those routines which are dominant, and finally we parallelise the code.

During the serial optimisation, we re-profile the code at the start of each phase, ensuring that the step about to be undertaken is the logically correct step to take.

During the parallelisation phase, the timings reported are the elapsed time on the wall clock, allowing the processing rate of polices per second, and the resulting speedup, to be intuitively interpreted by users in the business; the relationship between wall times and number of cores is readily visible.
2) Program Structure, Initial Profiling and Compiler Flags

In Section 1.3, we alluded to this investigation optimising some F90 written specifically for this project, rather than the F77 produced within the business. In this Chapter we properly introduce the code used in this investigation, its format and how it differs from the code produced by the valuation system used in the business.

Next we discuss the profiling used in this investigation, and consider the advantages and disadvantages of the technique used. We consider the profile of the initial code to highlight the possible routines which could be optimised, and discuss profiling of i/o.

We then consider the effect that different compiler options have on the code. The compiler used in this investigation has a wide range of options which can be set. However, at this stage of the investigation, we only consider flags which are likely to have a significant effect on performance, and have therefore not investigated those options which are not primarily concerned with enhancing performance, for example /names:lowercase (change routine names to all lowercase) etc. have been ignored.

Finally, we discuss some elementary changes to the code which are primarily of a software engineering benefit, and are designed to simplify later stages of optimisation, rather than enhance performance.

2.1) Structure of the Original Code

Before we start to optimise the code, we need to understand its structure. In this Section we introduce the F77 code produced by the valuation system, and discuss the differences between that F77 and the F90 code investigated in this project.

2.1.1) The Valuation System

In Section 1.2, we introduced the fact that the valuation package allows the user to enter relationships between variables, and that the package will transform those relationships into the source code of a conventional program. Within the valuation package, each variable must be entered separately, using a special form, which can be illustrated as:

![Figure 2.1: Imitation of the ‘Variable Entry’ screen in a valuation package](image-url)
Figure 2.1 represents the conditional relationships which are entered into the valuation package for the variable which calculates the cumulative ‘present value of retained earnings’ (PVRE). This is the cumulative value in the usual sense so that, at the start of the projection of each policy, the cumulative value is zero, and the PVRE in each step is added as the projection progresses.

An alternative interpretation is that, in the first step the cumulative PVRE is the PVRE for that step, and then, in all subsequent steps, the cumulative PVRE is the sum of its previous value and the PVRE for the current step. It must be noted that there is a requirement, which is fundamental to the workings of the package, that all references to all variables are be taken as the value as at the current step. Therefore, in order that the lexical analyser within the valuation package does not report a circular reference, the relationship must be entered as shown in Figure 2.1. Therefore, with the ‘_1’ interpreted as ‘the value of this variable at the end of the previous step’, it becomes apparent that the relationships given in Figure 2.1 will give the correct accumulation for PVRE, assuming that all other relationships in the package are correct.

When the user requires the program to be generated and run, the valuation package transforms the relationships into standard code. Continuing the example for the cumulative PVRE, the Fortran77 produced would be similar to;

```
if ( FirstTorTle0.eq.1 ) then
    CumulativePVRE = PVRE
else
    CumulativePVRE = CumulativePVRE_1 + PVRE
endif
```

Extract 2.1: Code for ‘cumulative PVRE’ produced by valuation package

Most valuation packages actually require that each relationship is specified separately. Therefore, if there are any conditions which are common to several relationships, then they need to be expressed within each of the variables which share that commonality.

Consider, for example, a variable for the cumulative ‘retained earnings’ (RE). This is extremely similar to the cumulative PVRE above, with the difference being the absence of allowance for interest earned between the valuation date and the time to which the retained earnings relate. The Fortran77 produced for this variable could be;

```
if ( FirstTorTle0.eq.1 ) then
    CumulativeRE = RE
else
    CumulativeRE = CumulativeRE_1 + RE
endif
```

Extract 2.2: Code for ‘cumulative RE’ produced by valuation package

By considering these two variables, it is immediately apparent that the code produced will have each of the variables within its own if-block, with vast scope for repetition of conditional tests. Such repetition, and the subsequent additional time required by the compiler, is clearly a disadvantage of these valuation packages.

2.1.2) The Code in this Investigation

The actual code produced by the valuation package has some fundamental issues which prevent its use in this report. The most prominent of these is that the code is of commercial interest, and so company policy forbids its use for anything other than the purposes for which it was originally intended, i.e. assisting in running the business.

The code investigated in this project is therefore an emulation of the F77 produced by the valuation package; it is based on an F77 structure using F90 syntax. The code from the valuation package would have a small main routine whose sole purpose is to call a routine, of roughly 3000 lines, which actually performs the required projections. However, as mentioned in Section 1.3, we have difficulty in producing large procedural programs, and so the F90 investigated is broken down into several subroutines which each have a coherent functional purpose.

Ignoring the actual division of the code into subroutines, the F90 we start from does still retain the features of the F77 produced by the package. For example, the repetition of the conditional tests, and the ‘_1’ variables for the values at the end of the previous step, are common throughout our code.
2.2) Profiling

This report covers the investigation into the performance of a Fortran code, and the subsequent improvement of that code. The standard starting point for such investigations is to obtain a profile of the supplied code, and base the investigation on that profile.

If the program will finally be run on a different platform to the development environment then it is sound practice to perform profiling on the system where it will ultimately be used, rather than the system on which development takes place. However, at the outset of this investigation no profiler on a multi-core system running Windows XP was available to this project, and so it was clearly not possible to profile the original code on the final system.

EPCC’s Ness is a multi-core system which will be used as an alternative system on which scalability of the parallel features of the final program will be gauged; this will be considered in Chapter 7. We therefore decided that, to ensure consistency, all profiling would be performed on Ness with the implicit assumption that optimisation based on the profile obtained on one platform would benefit performance on all platforms.

The profiling in this investigation was produced by a standard program-counter sampler, and several features of these samplers need to be highlighted in order that the results obtained from them are not misinterpreted by readers from the commercial environment, who are not conversant with software optimisation.

Firstly, while it may seem obvious, these profilers are samplers, and as such take information from several sampled points during the program’s execution. As a result of this sampling, it is likely that two identical runs of a particular program will have been sampled at different points, and so it is not possible to guarantee two such runs will produce the same profiling output. This lack of reproducibility leads to the inevitable conclusion that inferences about a program’s performance are dependent on the sampling achieved, and so any such inferences are not guaranteed to be correct; they are at best indications.

Secondly, the nature of sampling means that it is not possible to guarantee that all routines will be sampled, and so it is possible that two profiles of an identical program may indicate different routines being used. However, if a significant amount of time is spent in a routine then the nature of the sampling should guarantee that at least one sample point occurs during the execution of that routine, thereby leading to the routine appearing in the profile. Even allowing for this, it is obvious that the longer a program runs, the more accurate the profile will be. Therefore, it is clearly sensible to profile the program while it processes a number of policies which is sufficiently large that the sampling obtained is representative of the true profile, whilst not being so large that the additional processing does not increase the accuracy of the profile obtained.

Lastly, sampling itself takes time. The nature of sampling means that some of the sampling points will occur during the execution of the routine which produces the profile. A consequence of this is that as optimisation progresses, the actual time spent in calculation routines reduces and so the proportion of time spent in the profiling routine increases, thereby suggesting that ‘efficient’ versions of the code seem to require a great deal of sampling. It is therefore essential that such clearly misleading interpretations are avoided.

2.2.1) The Initial Profile

Despite the shortcomings just highlighted, samplers remain one of the most common aids to improving performance. These samplers are also generally well-understood and relatively simple to use, and so have been used in this project to monitor the progress of our optimisation.

Allowing for the discussion above regarding accuracy of the profile, all profiling performed to discover the effect of changes made during this investigation have been performed while the program process 100 policies. This number is large enough to ensure that the profile produced is representative, while still being small enough to be not too excessive.
The following Extract is from the profile of the original code, and shows only those routines which constitute more than 1% of the time.

```
% cumulative  self     total
 time  seconds  seconds    calls  s/call  s/call name
48.68   25.64  25.64   408108  0.00    0.00 annuityev_annressl
28.07   40.42 14.78 115859788  0.00    0.00 annuityev_lxinterp4pts
 6.23   43.70   3.28  
 6.23   43.70   3.28  
 6.23   43.70   3.28  
 4.99   46.33   2.63  
 4.79   48.85   2.52  
 3.33   50.61   1.75  
```

Extract 2.3: Profile of original code

Extract 2.3 suggests that the majority of the time is spent in a small number of routines. Interestingly, one of the routines highlighted is _mcount2, which is inserted into the profile by the profiler [6], and so there is nothing we can do to reduce its prominence. Hence, by removing _mcount2 from the analysis and combining the execution times of all the smaller routines, i.e. those not shown in Extract 2.3, the profile may be illustrated as

![Initial Profile](image)

Figure 2.2: Initial profile of code under investigation

From Figure 2.2 it is obvious that the execution is dominated by a small number of routines, which may suggest that we may be able to extract some increase in performance by manually optimising those routines. This will be considered further in Chapters 3 and 4.

2.2.2) Profiling i/o

We also profiled the original code while it processed each of 25, 50, 200 and 500 policies to see whether i/o was significant. Each of these profiles showed that the routines which used more than 1% of the time were the same as those in Extract 2.3. Further, in all cases, the most used i/o routine was f90io_encode_fmt, which always accounted for less than 0.05% of the overall time. This indicates that irrespective of how many policies are processed, i/o does not appear as a significant proportion of the user time.

The insignificance of the i/o is confirmed by the fact that the program under investigation provides two runtimes; one excludes i/o, while the other includes i/o. These times, for 1000 policies on Ness, where the profiling was performed, are 546.100 sec excluding i/o, and 546.122 sec including i/o. Hence, the proportion of overall time in i/o is approximately $4 \times 10^{-5}$. It is therefore clear that, with the code in its current state, optimising i/o will not have a measurable effect on the runtime. We will see in Section 6.1.3 that, even as optimisation progresses, i/o remains a negligible part of the overall time. Therefore, optimisation of i/o is not considered further in this investigation.
2.3) Compiler Flags

In any investigation into performance of a code, the standard first step is to try to discover what effect compiler flags have on the code produced. In this Section we discuss why such options are not available in the commercial environment, and then we consider the effect compiler flags have on the code under investigation.

2.3.1) The Valuation System

In Section 1.2.2 we discussed some of the problems associated with commercial valuation systems. Perhaps the most fundamental issue is that, in such systems, it is generally not possible to tune compiler or linker optimisations, and so the performance of the programs produced depends on compiler settings included in the valuation package at the time it was developed. This means that, irrespective of how far from optimal those settings are, none can be changed without the package itself being altered by its creators.

This lack of ability to tune the programs produced by the commercial package presents a discontinuity in the logic of how the performance of such programs can be enhanced. Within the commercial package, tuning compiler options can only be done after some negotiation with the external creators of the packages, whereas this investigation follows the approach of looking at compiler flags as the first stage. This inconsistency, and its resultant effect on the usefulness of this investigation to the commercial environment will be discussed in Section 8.2.

2.3.2) The Code in this Investigation

As mentioned in Section 1.3, this investigation is based on F90 code which runs on a laptop after having been compiled using Intel Visual Fortran V10.1, which, for brevity, we refer to by the name of its primary executable file, iFort. The fact that the full functionality of the compiler is available to this code enables us to perform our investigation to an arbitrary depth.

The iFort compiler is an extremely powerful tool, and as a result there are very many compiler flags. For current purposes, we only investigated those flags which looked like they could have a clearly beneficial effect on the performance of this code. The following Extract shows that part of the output from “ifort /help” which contains options which relate to basic optimisation:

```
Optimization
------------
/O1    optimize for maximum speed, but disable some optimizations which increase code size for a small speed benefit
/O2    optimize for maximum speed (same as /Ox)
/O3    enable /O2 plus more aggressive optimizations that may not improve performance for all programs
/Ox    enable maximum optimizations (DEFAULT)
/Od    disable optimizations; useful for selective optimizations (i.e. /Od /Og)
/fast  enable /QxT /O3 /Qipo /Qprec-div-
Options set by /fast cannot be overridden, list options separately to change behavior
/Ob<n> control inline expansion:
    n=0  disable inlining (same as /inline:none)
    n=1  no inlining (unless /Qip specified)
    n=2  inline any function, at the compiler's discretion (same as /Qip or /inline:all)
/Og[-] enable global optimizations
```

Extract 2.4: iFort’s basic optimisation flags
The runtimes for various combinations of these flags are

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
<th>Runtime (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>original settings [i.e. default optimisation]</td>
<td>987.9</td>
</tr>
<tr>
<td>01\a</td>
<td>set /O2 [maximise speed]</td>
<td>987.8</td>
</tr>
<tr>
<td>01\b</td>
<td>add /Ob2 [inline any suitable]</td>
<td>987.8</td>
</tr>
<tr>
<td>01\c</td>
<td>change to /Ob1 [no inlining unless /Qip specified]</td>
<td>406.7</td>
</tr>
<tr>
<td>01\d</td>
<td>add /unroll [let the compiler decide number of levels]</td>
<td>406.7</td>
</tr>
<tr>
<td>01\e</td>
<td>change from /O2 to /O3 [maximise speed plus higher]</td>
<td>406.7</td>
</tr>
<tr>
<td>01\f</td>
<td>add /Og [global optimisations]</td>
<td>406.7</td>
</tr>
<tr>
<td>01\g</td>
<td>add /fast</td>
<td>266.6</td>
</tr>
<tr>
<td>01\h</td>
<td>go straight from default to /fast [i.e. remove all others]</td>
<td>266.6</td>
</tr>
</tbody>
</table>

Table 2.1: Time to process 1000 policies with various compiler options

From the timings in Table 2.1, we can draw several conclusions:
1) The compiler’s default optimisation level includes the same set of transformations as are included in /O2; this is clearly consistent with the information from the manual, as shown in Extract 2.4.

2) /fast includes either greater global optimisations than /Og, or a selection of options which are more beneficial than those included in /Og.

3) Adding /fast to a combination of the other flags considered here produces the same runtime as using only /fast, and so it is justified to just use /fast.

Based on these conclusions, we deduce that /fast produces sufficiently efficient code that we use /fast for the remainder of this investigation.

2.3.3) Investigating the Components of /fast

In Section 2.3.2 we concluded that using only /fast is as beneficial as any other combination of flags tried. It was therefore considered useful to discover whether any particular aspect of /fast was more beneficial than any other.

From the part of the compiler manual shown in Extract 2.4, we see that /fast is made up of 4 components; /O3, /Qipo, /Qprec-div- and /QxT, which may be labelled, with descriptions, as

- a /O3 = high level serial optimisations
- b /Qipo = inter-procedural optimisations
- c /Qprec-div- = reduction in precision of division
- d /QxT = SSE extensions

Since there are 4 components, the total number of possible combinations of them is

$$\sum_{k=0}^{4} 4C_k = 4^4 = 16$$

and these combinations may be enumerated as

<table>
<thead>
<tr>
<th>Number of items</th>
<th>Number of combinations</th>
<th>Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4C0 = 1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4C1 = 4</td>
<td>a, b, c, d</td>
</tr>
<tr>
<td>2</td>
<td>4C2 = 6</td>
<td>ab, ac, ad, bc, bd, cd</td>
</tr>
<tr>
<td>3</td>
<td>4C3 = 4</td>
<td>abc, abd, acd, bcd</td>
</tr>
<tr>
<td>4</td>
<td>4C4 = 1</td>
<td>abcd</td>
</tr>
</tbody>
</table>

Table 2.2: Combinations for components of /fast
Re-running the code with each of these permutations of flags produces actual runtimes of

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
<th>Runtime (sec)</th>
<th>Fraction of “no components”</th>
</tr>
</thead>
<tbody>
<tr>
<td>01\k01</td>
<td>no components of /fast =&gt; default level of optimisation</td>
<td>987.8</td>
<td></td>
</tr>
<tr>
<td>01\k02</td>
<td>/O3</td>
<td>331.3</td>
<td>33.5%</td>
</tr>
<tr>
<td>01\k03</td>
<td>/Qipo</td>
<td>987.7</td>
<td>100.0%</td>
</tr>
<tr>
<td>01\k04</td>
<td>/Qprec-div-</td>
<td>978.2</td>
<td>99.0%</td>
</tr>
<tr>
<td>01\k05</td>
<td>/QxT</td>
<td>839.9</td>
<td>85.0%</td>
</tr>
<tr>
<td>01\k06</td>
<td>/O3 /Qipo</td>
<td>390.1</td>
<td>39.5%</td>
</tr>
<tr>
<td>01\k07</td>
<td>/O3 /Qprec-div-</td>
<td>327.2</td>
<td>33.1%</td>
</tr>
<tr>
<td>01\k08</td>
<td>/O3 /QxT</td>
<td>286.6</td>
<td>29.0%</td>
</tr>
<tr>
<td>01\k09</td>
<td>/Qipo /Qprec-div-</td>
<td>978.6</td>
<td>99.1%</td>
</tr>
<tr>
<td>01\k10</td>
<td>/Qipo /QxT</td>
<td>839.8</td>
<td>85.0%</td>
</tr>
<tr>
<td>01\k11</td>
<td>/Qprec-div- /QxT</td>
<td>832.4</td>
<td>84.3%</td>
</tr>
<tr>
<td>01\k12</td>
<td>/O3 /Qipo /Qprec-div-</td>
<td>392.8</td>
<td>39.8%</td>
</tr>
<tr>
<td>01\k13</td>
<td>/O3 /Qipo /QxT</td>
<td>282.9</td>
<td>28.6%</td>
</tr>
<tr>
<td>01\k14</td>
<td>/O3 /Qprec-div- /QxT</td>
<td>264.5</td>
<td>26.8%</td>
</tr>
<tr>
<td>01\k15</td>
<td>/Qipo /Qprec-div- /QxT</td>
<td>832.5</td>
<td>84.3%</td>
</tr>
<tr>
<td>01\k16</td>
<td>/O3 /Qipo /Qprec-div- /QxT</td>
<td>266.8</td>
<td>27.0%</td>
</tr>
</tbody>
</table>

Table 2.3: Time to process 1000 policies using various combinations of components of /fast

From the results in Table 2.3, we can make several observations:
1) taking the components one-at-a-time;
   a) /O3 has a significant effect, reducing the runtime to 33.5% of the time using default settings.
   b) /Qipo has no discernible effect.
   c) /Qprec-div- has almost no effect.
   d) /QxT has a minor effect; the runtime is reduced to 85.0% of the time using default settings.
2) taking the components two-at-a-time;
   a) /O3 with any other has effect of reducing to about 35% of the time using default settings.
   b) /O3 with /QxT brings the runtime to 29% of the time using default settings.
   n.b. 29% ≈ 33.5%×85%, so the effect of combining /O3 and /QxT seems linear.
3) taking the components three-at-a-time;
   a) /O3 and /QxT, with either of the other two, reduces the time to roughly 28% of the time using
      the default setting. This is consistent with both the lack of benefit from /Qipo and
      /Qprec-div-, and the linear effect of combining /O3 and /QxT.

From these observations, we can draw several conclusions:
1) the major gain is from /O3 together with /QxT.
2) the other components each have extremely minor effects.
3) we could just use /O3 with /QxT, but /fast is equally simple to understand, and has slightly greater
   benefit.

These results reinforce the conclusion, in Section 2.3.2, that the remainder of this investigation which is
performed on the laptop should use compilation with /fast.

2.4) Initial Simplification of the Code

In this Section we consider some simple changes to the code, which are intended primarily to make the code
easier to understand, change and use. As such, these changes are for reasons of software engineering, rather
than HPC, and are therefore not expected to have any significant effect on performance.

2.4.1) Debugging Mode vs. Live Mode

In an attempt to match the speed at which the code produced by the valuation package runs, the original version
of the code under investigation was written rather naively. As a direct result, checking whether full test output
is required is somewhat clumsy in that it uses the boolean variable isDebug which is hard-coded only once, and
then checked in several other places within the program.
This may initially seem inefficient, but the compiler should have the ability to produce code which replaces fixed-valued variables with the known values of those variables, i.e. the compiler should perform “constant folding” [7]. This naivety was removed by replacing the boolean variable with a pre-processor variable and invoking \( \text{fpp} \) at the compilation stage.

2.4.2) Variable Declarations

The original version of the live code emulates the valuation package, and within that package all variables are assigned to double precision real variables, which automatically results in all variables having a declared type before they are used. However, the code used in this investigation is written using F90 and so things are not so clear-cut. Therefore, to guarantee that all variables have been given the proper type, ‘\text{implicit none}’ was added to the program.

2.4.3) Number of Polices Processed

This code is designed to find the profitability within a certain number of policies, with that number being specified at some point within the program. For the early stages of this investigation, runs which are used to test correctness of changes process 100 policies, and runs to measure the code’s performance process 1000 policies. To simplify the process of changing between types of runs, the variable which dictates how many policies are processed was replaced by a pre-processor variable.

2.4.4) Effect on Performance

All of the changes mentioned above were implemented separately, and the actual runtimes for each of these changes were obtained. However, there was no measurable difference between the timings for these runs.

Irrespective of their effect on performance, all these changes were beneficial because they had the effect of improving readability, and maintainability, of the code.

2.5) Summary

In this Chapter we have introduced the code used in the investigation, and looked at its profile before any form of optimisation was done. After investigating which compiler flags were beneficial, we concluded that \( /\text{fast} \) should be used, and then looked into which components of \( /\text{fast} \) contributed to the performance increase. Finally we performed some changes which were beneficial in terms of software engineering.

Our investigation into compiler flags initially considered the three areas of optimisation level, inlining and loop unrolling. Having concluded that \( /\text{fast} \) had the greatest effect on performance, we performed an exhaustive search of which of its components were responsible for the performance gain.

In the initial stage of this phase of the investigation, we only looked at the combination of a few of the compiler’s flags. Tools now exist which use, for example, genetic algorithms to exhaustively search combinations of larger numbers of flags to see which combination is most beneficial. Any future work on the optimisation of code in this project could use one of these tools to see whether any better combination of compiler flags can be found.
3) Manual Optimisation of the Interpolation Routine

In Section 2.2.1, we saw that there were two routines which, between them, accounted for over 75% of the execution time. In this Chapter we consider the interpolation routine.

Following the optimisation of the compiler flags in Chapter 2, we re-profile the code in order to ensure that we are proceeding with the investigation in the correct order by optimising the most heavily used routines first.

Next, we consider application of the interpolation routine within the commercial environment, and its actual implementation in this program.

We then discuss the various changes made to the code during this stage of the optimisation. We performed several iterations, each of which made a small change to the code, and then timed the code while it processed 1000 policies. These timings, and any changes to the code’s performance are discussed after all the changes have been described.

During this stage of the investigation, we discovered that reordering some statements within the routine had a major effect on performance. This was an unexpected phenomenon, and so we discuss the performance of various permutations of the order of those statements.

3.1) Re-profiling the Code

In Chapter 2 we found compiler flags which improved performance, and we also made some changes to the code with the aim of simplifying it. It was therefore necessary to re-profile the code in order to establish the effect of those changes.

The following Extract shows the profile of the updated code as it runs after having been compiled with /fast. To be consistent with Extract 2.3, the following only shows those routines which constitute more than 1% of the execution time.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>cumulative</th>
<th>self seconds</th>
<th>seconds calls</th>
<th>s/call</th>
<th>total s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>40.31</td>
<td>40.31</td>
<td>9.63</td>
<td>9.63</td>
<td>0.00</td>
<td>0.00</td>
<td>annuityev lxinterp4pts</td>
</tr>
<tr>
<td>23.52</td>
<td>63.83</td>
<td>15.25</td>
<td>5.62</td>
<td>0.00</td>
<td>0.00</td>
<td>annuityev annressl</td>
</tr>
<tr>
<td>11.51</td>
<td>75.34</td>
<td>18.00</td>
<td>2.75</td>
<td>0.00</td>
<td>0.00</td>
<td>__fvdlog_long</td>
</tr>
<tr>
<td>9.34</td>
<td>84.68</td>
<td>20.23</td>
<td>2.23</td>
<td>0.00</td>
<td>0.00</td>
<td>_mcount2</td>
</tr>
<tr>
<td>5.76</td>
<td>90.44</td>
<td>21.61</td>
<td>1.38</td>
<td>0.00</td>
<td>0.00</td>
<td>__fvdexp_long</td>
</tr>
<tr>
<td>3.55</td>
<td>94.00</td>
<td>22.41</td>
<td>0.80</td>
<td>0.00</td>
<td>0.00</td>
<td>__fvdpow</td>
</tr>
<tr>
<td>1.67</td>
<td>95.67</td>
<td>22.80</td>
<td>0.40</td>
<td>0.00</td>
<td>0.00</td>
<td>__rouexit</td>
</tr>
<tr>
<td>1.12</td>
<td>96.79</td>
<td>23.07</td>
<td>0.27</td>
<td>0.00</td>
<td>0.00</td>
<td>f90io_close</td>
</tr>
<tr>
<td>1.09</td>
<td>97.88</td>
<td>23.34</td>
<td>0.26</td>
<td>0.00</td>
<td>0.00</td>
<td>annuityev maxdate</td>
</tr>
</tbody>
</table>

Extract 3.1: Profile after tuning compiler optimisations and initial simplification

Extract 3.1 highlights the benefit of good compiler optimisations; simply changing compiler flags caused the cumulative time for these dominant routines to fall to just under half its previous value.

By comparing Extract 3.1 with Extract 2.3, we see that the interpolation routine now constitutes more of the execution time than the reserving calculation, and this change of ordering suggests that the compiler has done a better job of optimising the calculation of reserves than the interpolation routine. There is, however, a possibility that the compiler has optimised both routines equally well, and that more time is spent in the interpolation routine because it is called many more times than the reserving calculation.

By repeating the process of removing _mcount2 from the analysis and combining the remaining routines so that those shown are consistent with Figure 2.2, the profile may be illustrated as
The fact that roughly 45% of the overall runtime is now spent in the interpolation routine shows that it should be the first routine to be investigated with a view to its optimisation. Also, Extract 3.1 shows that the routine is called $115.86 \times 10^6$ times during the profiling of these 100 policies, and so any performance benefit we can obtain from manually optimising this routine will have its effect amplified in the commercial environment where we process 100,000 policies in one dataset.

### 3.2) Use of the Interpolation Function

To guarantee that a smooth progression of mortality rates appears in the calculations, the program in the commercial environment uses a user-function, as described in Section 1.2.1, to perform interpolation within the $l_1$'s of the mortality table used to estimate survival probabilities [8].

Within the commercial environment, it has already been recognised that Gaussian elimination is inefficient as a means of finding a co-locating polynomial. Therefore, Lagrangian interpolation is used to fit a cubic polynomial through the points which symmetrically surround the required value of $x$, and we note that cubic, rather than linear, interpolation is used for business reasons.

The implementation of Lagrange’s formula used in the commercial program follows a standard text [9]. We are fitting a co-locating cubic polynomial through a set of 4 points $\{(x_i, y_i): i = 0, 1, 2, 3\}$, and so the calculation we require, in its most recognisable form, is

$$P_3(x) = \frac{x-x_1}{x_0-x_1} \frac{x-x_2}{x_0-x_2} \frac{x-x_3}{x_0-x_3} y_0 + \frac{x-x_0}{x_1-x_0} \frac{x-x_2}{x_1-x_2} \frac{x-x_3}{x_1-x_3} y_1 + \frac{x-x_0}{x_2-x_0} \frac{x-x_1}{x_2-x_1} \frac{x-x_3}{x_2-x_3} y_2 + \frac{x-x_0}{x_3-x_0} \frac{x-x_1}{x_3-x_1} \frac{x-x_2}{x_3-x_2} y_3$$  \hspace{1cm} (3.1)

In order to implement this calculation, the function written to be used in the valuation package is based on an alternative representation of the formula, viz;

$$P_3(x) = L_0(x)y_0 + L_1(x)y_1 + L_2(x)y_2 + L_3(x)y_3 = \sum_{k=0}^{3} L_k(x) \cdot y_k \quad \text{where} \quad L_k(x) = \prod_{m=0}^{3} \frac{x-x_m}{x_k-x_m} \hspace{1cm} (3.2)$$
This interpolation is used to find \( l_x \) at non-integer ages. For illustration, suppose we require \( l_{71.30} \) from the AM92 life table. This can be represented graphically as

\[
\begin{align*}
\text{Figure 3.2: Example of interpolation in } l_x
\end{align*}
\]

From Figure 3.2, it is clear that the commercial strategy of fitting the cubic through the four points which correspond to the two integer ages either side of the required fractional value should provide an extremely accurate result. So, for this example, to obtain \( l_{71.30} \) we would use \( x = \{70, 71, 72, 73\} \). More generally, to find \( l_x \) for a general \( x \), we use the points corresponding to

\[
\{\lfloor x-1 \rfloor, \lfloor x \rfloor, \lfloor x+1 \rfloor, \lfloor x+2 \rfloor \}
\]

where \( \lfloor y \rfloor = \text{floor}(y) \).

Therefore, in the code under investigation, the original implementation of the interpolation routine is

```
double precision function lxInterp4pts(actualAge, lxTable)
  double precision actualAge, lxTable(0:121)
  integer k, m !loop counters
  integer baseAge !arguments for calls to mortalityTable
  double precision age(0:3) !arguments for calls to mortalityTable
  double precision lx(0:3) !returns from mortalityTable
  double precision lagrangePoly(0:3) !the lagrangian factors

  baseAge = idint(actualAge)-1
  do k=0,3
    age(k) = baseAge + k
    lx(k) = lxTable(age(k))
  end do
  lagrangePoly(0:3)=1.0d0
  do k=0,3
    do m=0,3
      if (m.ne.k) then
        lagrangePoly(k) = lagrangePoly(k) * dble(actualAge-age(m)) / dble(age(k)-age(m))
      end if
    end do
  end do
  lxInterp4pts = dot_product(lagrangePoly,lx)
  return
end function lxInterp4pts
```

Extract 3.2: Original implementation of the interpolation function

Inspection of the code in Extract 3.2 suggests that there are several opportunities for improving performance, and these are considered next.
3.3) Optimising the Function

The overall time spent in this routine means that any optimisations should have a particularly beneficial effect on overall performance. In this Section we consider the changes made to this routine, while their effect on execution times is considered in Section 3.4.

3.3.1) Using Features of the F90 Language

Since we are modifying the code with a view to improving performance, and the code under investigation is F90, it seems sensible to see whether new features of F90 provide any performance benefit.

Declaring functions and subroutines as \textit{pure} forbids certain i/o functionality, and other side-effects such as modifying the values of the arguments passed into these methods. It is suggested that, in some circumstances, forbidding such activity may aid optimising compilers [10]. Since this is an investigation into performance, anything which aids the compiler is likely to be beneficial, and hence the first step in optimising this function was to declare it as \textit{pure}, and add the required \texttt{intent()} clauses.

Since we are moving towards using the features of F90, we changed all variables which were declared as \texttt{double precision} to be declared as \texttt{real(8)} as the first stage towards allowing relatively simple switching to \texttt{real(4)} in the future. Such switching of precision level would be performed by the introduction of a pre-processor variable, say \texttt{realPrecision}, which would be \texttt{#defined} as either 8 or 4. This would allow an investigation as to whether using lower precision has any measurable performance advantage.

However, that investigation is postponed to a different project because changing the precision level of the variables passed between this function and the rest of the program requires that many other variables be declared with different precision levels. Such re-declarations within the code were not performed here for two main reasons: firstly, there are likely to be larger performance gains which are more easily achieved than re-engineering some of the code and, secondly, that changing precision level may have the unwanted side-effect of altering the results output from the program.

In the original version of the code for this routine, the parameter \texttt{actualAge} is passed in as a \texttt{real}, and converted to an \texttt{integer} in the process of determining \texttt{baseAge}. However, \texttt{baseAge} is then immediately converted back to a set of \texttt{reals} in the calculation of \texttt{age(:)}. This recasting is clearly unnecessary, and so these conversions have been removed by declaring \texttt{baseAge} as \texttt{real}, and using \texttt{floor()} rather than \texttt{idint()}.

Another simple change which moves the code further towards F90 is the use of array syntax. Therefore, the short loops which initialise array values have been modified to use array syntax. This change is especially appealing since it may allow the compiler to perform further performance enhancing transformations, e.g. vectorisation when compiling for use with SSE instructions which, as we discovered in Section 2.3.3, are one of the beneficial components of \texttt{/fast}. The revised code for these initialisations is

\begin{verbatim}
baseAge = floor(actualAge - 1.0d0)
age(0:3) = (/ baseAge , baseAge+1.0d0 , baseAge+2.0d0 , baseAge+3.0d0 /)
Ix(0:3) = lxTable(age(0:3))
\end{verbatim}

Extract 3.3: Initialisation of arrays in the interpolation function

3.3.2) Unrolling the Loops

Unrolling loops is often a good source of performance gains, and so the nest of two loops which perform the main calculation of \texttt{lagrangePoly(:)} have been unrolled. By performing this unrolling, it became apparent that there is no need to have separate initialisations of \texttt{lagrangePoly(:)=1.0d0}, so these have also been incorporated into the calculation. Taking the array’s first element as the example, the code for the calculation of \texttt{lagrangePoly(0)} is now

\begin{verbatim}
lagrangePoly(0) = 1.0d0 * dble(actualAge-age(1)) / dble(age(0)-age(1)) &
                  * dble(actualAge-age(2)) / dble(age(0)-age(2)) &
                  * dble(actualAge-age(3)) / dble(age(0)-age(3))
\end{verbatim}

Extract 3.4: Fully unrolled calculation of lagrangePoly\{k\}
3.3.3) Cleaning the Code

By considering the calculation of \( \text{lagrangePoly}(k) \) together with the initialisations of the arrays, it became apparent that \((\text{age}(p) - \text{age}(q))\), for arbitrary \(p\) and \(q\), is actually some rearrangement of \(((\text{baseAge}+u) - (\text{baseAge}+v))\) where \(u\) and \(v\) are integer constants. It therefore seemed sensible to perform the algebra to find the required value of \((\text{age}(p) - \text{age}(q))\) and place the resulting constants in the code. This naturally led to the elimination of the array \(\text{age}(:)\), and also to the replacement, by calculations, of other references to the array.

Having replaced each of the occurrences of \((\text{age}(p) - \text{age}(q))\) by constants, it became clear that the overall divisor in the calculation of each \(\text{lagrangePoly}(k)\) is constant. When compared to multiplication, division is an expensive operation, and so these division-by-constant operations have been replaced with a multiply-by-reciprocal, where the reciprocal is declared as a pre-processor variable, which \text{fpp} will substitute into the code.

Taking the array’s first element as the example again, the code for the calculation of \(\text{lagrangePoly}(0)\) is now:

```c
#define ONE_SIXTH 0.16666666666666666666666d0
lagrangePoly(0) = - ONE_SIXTH * dble(actualAge-(baseAge+1.0d0)) &
    dble(actualAge-(baseAge+2.0d0)) &
    dble(actualAge-(baseAge+3.0d0))
```

**Extract 3.5: Simplification of unrolled calculation of \(\text{lagrangePoly}(k)\)**

It is now clear that \((\text{actualAge}-\text{baseAge})\) is to be evaluated several times within the calculation of each \(\text{lagrangePoly}(k)\). Extract 3.3 shows that \text{actualAge} and \text{baseAge} are actually related to each other by the \text{floor()} function. It is therefore apparent that the overall calculation could be simplified by introducing some new variable to represent the fractional part of the age given by \((\text{actualAge}-\text{baseAge})\). Introducing a further array to hold the results of adding the constants to the fractional part of the age, and changing the point where the “deduction of one” is made, means that the body of the code is now:

```c
ageFraction = actualAge - floor(actualAge)
ageAdjust(0:3) = (/ ageFraction + 1.0d0 &
    ageFraction &
    ageFraction - 1.0d0 &
    ageFraction - 2.0d0 &
    /
)
lagrangePoly(0) = - ONE_SIXTH * ageAdjust(1) * ageAdjust(2) * ageAdjust(3)
lagrangePoly(1) = ONE_HALF * ageAdjust(0) * ageAdjust(2) * ageAdjust(3)
lagrangePoly(2) = - ONE_HALF * ageAdjust(0) * ageAdjust(1) * ageAdjust(3)
lagrangePoly(3) = ONE_SIXTH * ageAdjust(0) * ageAdjust(1) * ageAdjust(2)
```

**Extract 3.6: Further simplification of unrolled calculation of \(\text{lagrangePoly}(k)\)**

There are still several minor issues which suggest that the code may not perform optimally. The first of these is the fact that the values in \(\text{lx}(:)\) are obtained from the array \(\text{lxTable}(:)\) using \text{real} indices rather than \text{integer} indices, which suggests that some unnecessary recasting may be being performed. This was avoided by adding an integer variable on which the array indices may be based.

Given the number of times this routine is called, any gain, however small, could have a measurable effect on the overall runtime. One of the major steps in this routine is to find the values of some of the elements of an array which is passed in. Clearly, array indices are integers, and so referencing array elements using double precision variables requires a recasting operation. We therefore used \text{F90’s idint()} function to combine this recasting operation with the operation to find the relevant integer age. Having done this, it became possible to adjust the integers used as indices to populate \(\text{lx}(:)\) as part of that population process. The code for this is:

```c
ageLastBdy = idint(actualAge)
x(0:3) = lxTable( (/ ageLastBdy-1, ageLastBdy, ageLastBdy+1, ageLastBdy+2 /) )
```

**Extract 3.7: Replacement of floating-point indices**

The next opportunity for simplification came from the fact that it seemed likely that the pre-processor variables in this function are likely to be of use in other routines, later in the project. They have therefore been moved out of this method and placed at the start of the program.

The last calculation of this function is to evaluate the \text{dot_product()} which represents \(P_j(x) = \sum_{k=0}^{3} L_k y_k\).

This \text{dot_product()} function is provided in \text{F90}, and so is a general purpose function which can be used for arrays of arbitrary length. Therefore, within the function there must be some means of determining the length of each of the arrays passed in as arguments, and checking that these lengths are equal. These pre-calculation
checks, together with the time to call the function, and return from it, add an overhead to the actual calculation. In order to remove these overheads, this sum has been re-implemented in the expanded form

\[ P_3(x) = L_0y_0 + L_1y_1 + L_2y_2 + L_3y_3 \]

### 3.3.4) Permutations of Initialisation Statements

The code is now in the state where it seems that not much more performance can be squeezed out of changing the actual calculations. However, the initialisation statements and the calculation of the lagrangian factors are

\[
\begin{align*}
\text{ageLastBdy} &= \text{idint} (\text{actualAge}) \\
\text{ageFraction} &= \text{actualAge} - \text{floor} (\text{actualAge}) \\
\text{lx}(0:3) &= \text{lxTable} ( (\text{ageLastBdy}-1, \text{ageLastBdy}, \text{ageLastBdy+1}, \text{ageLastBdy+2} ) ) \\
\text{ageAdjust}(0:3) &= (\text{ageFraction}+1.0d0, \text{ageFraction}, \text{ageFraction}-1.0d0, \text{ageFraction}-2.0d0) \\
\text{lagrangePoly}(0) &= -\frac{1}{6} \cdot \text{ageAdjust}(1) \cdot \text{ageAdjust}(2) \cdot \text{ageAdjust}(3) \\
\text{lagrangePoly}(1) &= \frac{1}{2} \cdot \text{ageAdjust}(0) \cdot \text{ageAdjust}(2) \cdot \text{ageAdjust}(3) \\
\text{lagrangePoly}(2) &= -\frac{1}{2} \cdot \text{ageAdjust}(0) \cdot \text{ageAdjust}(1) \cdot \text{ageAdjust}(3) \\
\text{lagrangePoly}(3) &= \frac{1}{6} \cdot \text{ageAdjust}(0) \cdot \text{ageAdjust}(1) \cdot \text{ageAdjust}(2)
\end{align*}
\]

From Extract 3.8, it is apparent that the lagrangian factors, \text{lagrangePoly}(:)\), depend on the values of the \text{ageAdjust}(:)\) array from the immediately preceding calculation. We therefore decided to try reordering some of these statements to see whether any performance gains could be found. The simplest and most obvious rearrangement, which allows some possibility that \text{ageAdjust}(:)\) may be fully calculated before it is required in \text{lagrangePoly}(0)\), is obtained by swapping the initialisation of the two arrays, viz;

\[
\begin{align*}
\text{ageAdjust}(0:3) &= (\text{ageFraction}+1.0d0, \text{ageFraction}, \text{ageFraction}-1.0d0, \text{ageFraction}-2.0d0) \\
\text{lx}(0:3) &= \text{lxTable} ( (\text{ageLastBdy}-1, \text{ageLastBdy}, \text{ageLastBdy+1}, \text{ageLastBdy+2} ) )
\end{align*}
\]

Extract 3.9: Rearranged initialisation statements

### 3.3.5) Speed of Light

Having reached this point, significant gains look difficult to find and so it seems sensible to see how far from optimum this version of the code is. Using the “speed of light” approach [11], we can estimate the best possible performance of this routine by calling a nonsense version which just returns a hard-coded number. Bearing in mind that this yields results which are meaningless in terms of their intended purpose, by using this method we can see how fast the program could run if all calculations in this routine were done ‘in no time’. Code for this test is

\[
\begin{align*}
\text{pure function lxInterp} & (\text{actualAge}, \text{lxTable}) \text{ result}(\text{reqdLX}) \\
\text{real}(8), \text{ intent(in) :: } & \text{actualAge}, \text{lxTable(0:121)} \\
\text{real}(8) :: & \text{reqdLX}
\end{align*}
\]

\[
\text{reqdLX} = 11523.33d0
\]

Extract 3.10: Interpolation at the speed-of-light
3.4) Effect on Performance

The actual runtimes for each stage of the optimisation are

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
<th>Runtime (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>03^a1</td>
<td>make lxInterp4pts pure {&amp; change its name}</td>
<td>265.9</td>
</tr>
<tr>
<td>03^a2</td>
<td>change double precision to real(8)</td>
<td>265.9</td>
</tr>
<tr>
<td>03^a3</td>
<td>change baseAge to floor(actualAge - 1.0d0)</td>
<td>266.7</td>
</tr>
<tr>
<td>03^a4</td>
<td>calculate age(:) and lx(:) using array syntax</td>
<td>266.6</td>
</tr>
<tr>
<td>03^b1</td>
<td>unroll outer loop</td>
<td>265.0</td>
</tr>
<tr>
<td>03^b2</td>
<td>unroll inner loop</td>
<td>260.9</td>
</tr>
<tr>
<td>03^b3</td>
<td>combine calculations for each of lagrangePoly(:)</td>
<td>228.5</td>
</tr>
<tr>
<td>03^c1</td>
<td>replace age(;) with simplification of (base+p)-(base+q)</td>
<td>188.4</td>
</tr>
<tr>
<td>03^c2</td>
<td>combine divisions into one</td>
<td>184.1</td>
</tr>
<tr>
<td>03^c3</td>
<td>change (/6.0) to (*{\text{ONE_SIXTH}})</td>
<td>184.9</td>
</tr>
<tr>
<td>03^c4a</td>
<td>change actualAge-baseAge to be based on fractional age</td>
<td>188.5</td>
</tr>
<tr>
<td>03^c4b</td>
<td>reformat the layout of the code {no changes to calculations}</td>
<td>188.6</td>
</tr>
<tr>
<td>03^c5</td>
<td>change lx(:) to lookup on integers {rather than doubles}</td>
<td>182.0</td>
</tr>
<tr>
<td>03^c6</td>
<td>change from agePrevBdy to ageLastBdy</td>
<td>186.2</td>
</tr>
<tr>
<td>03^c7</td>
<td>move preProcessor variables out of routine {to start of file}</td>
<td>186.1</td>
</tr>
<tr>
<td>03^c8</td>
<td>manually unwind the dot_product() function</td>
<td>181.8</td>
</tr>
<tr>
<td>03^d</td>
<td>reorder the initialisations</td>
<td>158.3</td>
</tr>
<tr>
<td>03^z</td>
<td>speed of light</td>
<td>155.7</td>
</tr>
</tbody>
</table>

Table 3.1: Time to process 1000 policies as optimisation of interpolation routine progressed

From the results in Table 3.1, we can make several observations;

1) the runtime before optimisation was about 266.8 sec, as shown in Table 2.3, and the runtime using the speed-of-light version of this routine was around 155.7 sec. Therefore, before optimisation, the time spent in the interpolation routine was around 266.8−155.7 = 111.1 sec. Similarly, the difference in runtimes between the final version of this routine and the speed-of-light version indicates that, after optimisation, the time spent in the interpolation routine was around 158.3−155.7 = 2.6 sec. We have therefore reduced the time spent in this routine by \(\left(1-\frac{2.6}{111.1}\right) \approx 97.6\%\). An alternative way to look at this is that we have improved the performance of this routine by a factor of \(\frac{111.1}{2.6} \approx 42.7\).

2) versions 3a1 to 3a4 indicate that changing the code to take advantage of F90 features has no real effect on the runtime. Irrespective of their lack of effect on performance, these changes are worth making simply because they have the effect of improving the readability, and maintainability, of the code.

3) versions 3b1 to 3b3 indicate that unrolling the loops had a significant effect on the overall runtime, reducing it from 266.6 sec to 228.5 sec. This is a reduction of roughly 14.3\%, and is clearly consistent with the fact that unrolling loops is a beneficial thing to do.

4) version 3c1 indicates that simplifying calculations such as \(\text{dble}(\text{age}(0)-\text{age}(1))\) had a significant effect on runtime, reducing it from 228.5 sec to 188.4 sec. This is a reduction of roughly 17.6\%, and results from simply removing some of the convoluted arithmetic which was used.

5) versions 3c2 to 3c8 indicate that miscellaneous alterations to the code, to make it easier to read, do not have any major effect on the runtime. While some of the changes marginally improve performance, other changes have minor detrimental effects. However, even if a change slowed execution slightly it has been left in the code in order to maintain readability, and this is consistent with good software engineering practice.

6) version 3d indicates that reordering initialisations had a significant effect on runtime, reducing it from 181.8 sec to 158.3 sec, is a reduction of about 12.9\%. Since we did not expect a performance gain of this magnitude, the effect of ordering of initialisations is considered further in Section 3.5.
7) version 3z indicates that if this method ran at the ‘speed of light’ then the runtime for the overall program would be 155.7 seconds, which is only 1.6% faster than the current optimised version. This suggests that, allowing for the fact that some calculations need to be performed, the last version is not too far from optimal. Certainly, it is hard to see where any further improvements could be made.

Although we have been through several stages of optimisation, and improved the performance of this routine by a factor in excess of 40, it is apparent that there are only three really beneficial steps:

1) loop unrolling
2) simplification of arithmetical steps
3) reordering statements.

3.5) Investigating the Permutations of Initialisation Statements

In Section 3.3.4 we swapped the order of two of the initialisation statements, and in Section 3.4 we saw that this swap had a significant effect on runtime. This reduction in runtime led to an investigation into whether the new ordering of statements was actually the optimal ordering.

3.5.1) Possible Permutations

The full code for the interpolation routine, with the statements in the order they naturally appeared before we reordered some of them in Section 3.3.4 is

```fortran
pure function lxInterp(actualAge, lxTable) result(reqdLX)
  integer :: ageLastBdy  ! age at previous birthday
  real(8) :: ageFraction
  real(8), intent(in) :: actualAge, lxTable(0:121)
  real(8) :: reqdLX

  ageLastBdy = idint(actualAge)
  ageFraction = actualAge - floor(actualAge)
  lx(0:3) = lxTable( (/ ageLastBdy-1 , ageLastBdy, ageLastBdy+1, ageLastBdy+2 /) )
  ageAdjust(0:3) = (/ ageFraction+1.0d0, ageFraction, ageFraction-1.0d0, ageFraction-2.0d0 /)

  lagrangePoly(0) = - ONE_SIXTH * ageAdjust(1) * ageAdjust(2) * ageAdjust(3)
  lagrangePoly(1) =   ONE_HALF  * ageAdjust(0) * ageAdjust(2) * ageAdjust(3)
  lagrangePoly(2) = - ONE_HALF  * ageAdjust(0) * ageAdjust(1) * ageAdjust(3)
  lagrangePoly(3) =   ONE_SIXTH * ageAdjust(0) * ageAdjust(1) * ageAdjust(2)

  reqdLX = lagrangePoly(0)*lx(0) + lagrangePoly(1)*lx(1) + lagrangePoly(2)*lx(2) + lagrangePoly(3)*lx(3)
end function lxInterp
```

Extract 3.11: The original version of the optimised interpolation routine

It is clear from Extract 3.11 that the final calculation is the evaluation of the dot-product, and so there is no reason why all previous statements and assignments in the calculation of lagrangePoly(:) cannot be regarded as part of the initialisation of that dot-product.

For clarity, we label the first four initialisation statements as a to d, and the calculation of lagrangePoly(:) as e, viz;

a  ageLastBdy = idint(actualAge)
b  ageFraction = actualAge - floor(actualAge)
c  lx(0:3) = lxTable( (/ ageLastBdy-1 , ageLastBdy, ageLastBdy+1, ageLastBdy+2 /) )
d  ageAdjust(0:3) = (/ ageFraction+1.0d0, ageFraction, ageFraction-1.0d0, ageFraction-2.0d0 /)
e  lagrangePoly(0) = - ONE_SIXTH * ageAdjust(1) * ageAdjust(2) * ageAdjust(3)
lagrangePoly(1) =   ONE_HALF  * ageAdjust(0) * ageAdjust(2) * ageAdjust(3)
lagrangePoly(2) = - ONE_HALF  * ageAdjust(0) * ageAdjust(1) * ageAdjust(3)
lagrangePoly(3) =   ONE_SIXTH * ageAdjust(0) * ageAdjust(1) * ageAdjust(2)

Extract 3.12: Labelled statements in the body of the interpolation routine
With this labelling, it is apparent that \( a \) must appear before \( c \), and also that \( b, d \) and \( e \) can only appear in that order. This means that the statements can appear in any order so long as the restrictions that the orderings \( ac \) and \( dbe \) are observed. The number of possible orderings is therefore

\[
\frac{5!}{2! \cdot 3!} = 10
\]

and these orderings may be enumerated as

\[
\{abcde, abdce, abdec, acbde, bacde, badce, bdaec, bdeac, bdace, bdeac\}
\]

Since there are only ten allowable permutations of these statements, and they are not too onerous to code, all have been included in the analysis. The actual runtimes for each of the possible permutations is

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
<th>Runtime (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>03(f01)</td>
<td>statements ordered abcde</td>
<td>181.4</td>
</tr>
<tr>
<td>03(f02)</td>
<td>statements ordered abdce</td>
<td>158.3</td>
</tr>
<tr>
<td>03(f03)</td>
<td>statements ordered abdec</td>
<td>161.4</td>
</tr>
<tr>
<td>03(f04)</td>
<td>statements ordered acbde</td>
<td>186.6</td>
</tr>
<tr>
<td>03(f05)</td>
<td>statements ordered bacde</td>
<td>182.7</td>
</tr>
<tr>
<td>03(f06)</td>
<td>statements ordered badce</td>
<td>158.0</td>
</tr>
<tr>
<td>03(f07)</td>
<td>statements ordered badec</td>
<td>162.5</td>
</tr>
<tr>
<td>03(f08)</td>
<td>statements ordered bdace</td>
<td>158.1</td>
</tr>
<tr>
<td>03(f09)</td>
<td>statements ordered bdaec</td>
<td>163.0</td>
</tr>
<tr>
<td>03(f10)</td>
<td>statements ordered bdeac</td>
<td>163.4</td>
</tr>
</tbody>
</table>

Table 3.2: Time to process 1000 policies for permutations of statements in interpolation routine

From the results in Table 3.2, we can make several observations;

1) version \( 3f01 \) has a runtime which is almost identical to the runtime for \( 3c8 \), as given in Table 3.1, and this is a result of the two version having the same ordering, and hence the variation in this pair of times is within the variability that could reasonably be expected after allowing for the daemons on the system.

Similarly, version \( 3f06 \) has a runtime which is almost identical to the runtime for \( 3d \), as given in Table 3.1, which again results from the orderings being the same. Whilst these observations reveal nothing new about the performance of the code, they do add much to the confidence about the reproducibility of the results.

2) we can summarise the results in Table 3.2 into groups, as follows

<table>
<thead>
<tr>
<th>Runtime (sec)</th>
<th>Versions</th>
<th>Order [ignoring ( a ) and ( b )]</th>
</tr>
</thead>
<tbody>
<tr>
<td>181 – 186</td>
<td>03(f01), 03(f04), 03(f05)</td>
<td>c d e</td>
</tr>
<tr>
<td>161 – 163</td>
<td>03(f03), 03(f07), 03(f09), 03(f10)</td>
<td>d e c</td>
</tr>
<tr>
<td>157.9 – 158.4</td>
<td>03(f02), 03(f06), 03(f08)</td>
<td>d c e</td>
</tr>
</tbody>
</table>

Table 3.3: Grouping of timings for permutations in interpolation routine

Using these groupings, it is apparent that the ordering of \( a \) and \( b \) does not have a major effect on the runtime.

3) following on from 2, disregarding the order of \( a \) and \( b \) leads to the three distinct permutations of the remaining statements, i.e. \{cde, dce, dec\}. The timings for these groupings confirm that, in order to obtain fast execution of the code, \( d \) should precede \( c \), and that the fastest executions are obtained where \( e \) follows \( d \) and \( c \).

4) the permutation \( bdace \) has the shortest runtime, albeit only marginally, indicating that to be the optimum ordering.

These observations indicate that the ordering of the statements which leads to the fastest possible execution is

\[
\begin{align*}
\text{ageFraction} &= \text{actualAge} - \text{floor(actualAge)} \\
\text{ageLastBdy} &= \text{idint(actualAge)} \\
\text{ageAdjust}(0:3) &= (/ \text{ageFraction}+1.0d0, \text{ageFraction}, \text{ageFraction}-1.0d0, \text{ageFraction}-2.0d0 /) \\
\text{lx}(0:3) &= \text{lxTable}( (/ \text{ageLastBdy}+0, \text{ageLastBdy}, \text{ageLastBdy}+1, \text{ageLastBdy}+2 /) ) \\
\text{lagrangePoly}(0) &= \text{ONE_SIXTH} * \text{ageAdjust}(1) * \text{ageAdjust}(2) * \text{ageAdjust}(3) \\
\text{lagrangePoly}(1) &= \text{ONE_HALF} * \text{ageAdjust}(0) * \text{ageAdjust}(2) * \text{ageAdjust}(3) \\
\text{lagrangePoly}(2) &= \text{ONE_HALF} * \text{ageAdjust}(0) * \text{ageAdjust}(1) * \text{ageAdjust}(3) \\
\text{lagrangePoly}(3) &= \text{ONE_SIXTH} * \text{ageAdjust}(0) * \text{ageAdjust}(1) * \text{ageAdjust}(2)
\end{align*}
\]

Extract 3.13: Optimal ordering of statements in the interpolation routine
For the remainder of this investigation, we use the code given in Extract 3.13 as the optimal version of this function.

### 3.5.2) Future Investigation of the Optimal Permutation

Although we have discovered which ordering of statements in the interpolation routine gives optimal performance, we have not been able to identify exactly why the optimum ordering is so advantageous. Unfortunately, because this investigation has limited duration, there was insufficient time to pursue this particular avenue further.

In situations where features of performance are either unexpected or unexplained, there is generally a temptation to cite cache effects. If time had allowed, we could have investigated whether cache had any beneficial effect. A suggested approach for this would be to discover whether the layout of arrays were linked to the length of cache lines, and this could be achieved by using Fortran common blocks in order to force the memory to be laid out as required.

Whilst moving on to the next part of the investigation without finding the exact cause of the performance differences, Section 5.3.2 will consider the performance improvements resulting from optimisation of this interpolation routine on different platforms. There, it becomes apparent that the effect of ordering the statements differently is not so pronounced on those other platforms as it is on the laptop. It is therefore sensible not to allocate any more time to this investigation of, what turns out to be, a transient feature which is localised to a particular platform.

### 3.6) Summary

In this Chapter we have investigated the manual optimisation of the interpolation routine. We have performed many small changes and have discovered that those changes which had a major effect can be grouped into three categories:

1) loop unrolling
2) simplification of arithmetical steps
3) reordering statements.

The optimal version of the code is an implementation of a form of Lagrange’s formula which is algebraically equivalent to the form presented in Equation (3.1); further details of the algebra are presented in Appendix A. It should therefore have been possible to have performed the algebraic rearrangement before any implementation, so that the initial version of the code was the most optimal. While this conclusion is of future interest for software engineering projects, for our current investigation it is clearly an example of hindsight.

In Section 3.3.1, we highlighted the change to using real(8), but we did not pursue the change to single precision further. Since single precision arithmetic is performed in half the time of double precision arithmetic, it seems reasonable to think that changing to single precision would reduce the time spent in this routine still further. Discovering whether the reduction in execution time is significantly smaller than the current time would make a natural basis for a further investigation.
4) Manual Optimisation of the Reserve Calculation

In Section 2.2.1, we saw that there were two routines which, between them, accounted for over 75% of the execution time of the original code. In Chapter 3 we considered the interpolation algorithm and its optimisation. In this Chapter we consider the routine which calculates the reserves.

Proceeding as in Chapter 3, the first stage of this part of the investigation is to re-profile the code in order to ensure that manual optimisation of the reserving calculation is consistent with a logical approach to improving the performance of the program.

Next, we consider the derivation of the calculations required, and include some justification for the number of times the calculation is performed.

We then discuss the various changes made to the code during this stage of the optimisation. Again, to be consistent with Chapter 3, we performed several iterations, each of which made a small change to the code, and then timed the code while it processed 1000 policies. In order to continue with the same structure as Chapter 3, the timings, and any changes to the code’s performance, are discussed after all the changes to the code have been described.

During this stage of the investigation, we replaced some power calculations with a sequence of multiplication calculations, and that had a dramatic effect on performance. So, in order to ensure that the reduction in runtime was consistent with expectation, we look at the number of power calculations which have been replaced.

4.1) Re-profiling the Code

Section 2.2.1 highlighted the potential for two routines to be investigated. In Chapter 3 we optimised the interpolation algorithm, and that had a major impact on the runtime of the code. Therefore, before continuing, it was again necessary to re-profile the program to ensure that the second routine highlighted in Section 2.2 is still a logical candidate for optimisation.

The following Extract shows the profile of the updated code and, to maintain consistency with Extracts from previous profiles, only shows those routines which constitute more than 1% of the time.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>seconds</th>
<th>self</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>34.92</td>
<td>6.01</td>
<td>6.01</td>
<td></td>
<td></td>
<td>annuityev_annressl</td>
</tr>
<tr>
<td>17.65</td>
<td>9.05</td>
<td>3.04</td>
<td></td>
<td></td>
<td>__fvdllog_long</td>
</tr>
<tr>
<td>13.21</td>
<td>11.32</td>
<td>2.27</td>
<td></td>
<td></td>
<td>__mcount2</td>
</tr>
<tr>
<td>7.55</td>
<td>12.62</td>
<td>2.27</td>
<td></td>
<td></td>
<td>__fvdexp_long</td>
</tr>
<tr>
<td>6.89</td>
<td>13.81</td>
<td>1.30</td>
<td></td>
<td></td>
<td>annuityev_lxinterp</td>
</tr>
<tr>
<td>5.15</td>
<td>14.69</td>
<td>0.89</td>
<td></td>
<td></td>
<td>__fvdpow</td>
</tr>
<tr>
<td>2.60</td>
<td>15.14</td>
<td>0.45</td>
<td></td>
<td></td>
<td>__rouexit</td>
</tr>
<tr>
<td>1.96</td>
<td>15.48</td>
<td>0.34</td>
<td></td>
<td></td>
<td>pgf90_dfloor</td>
</tr>
<tr>
<td>1.96</td>
<td>15.82</td>
<td>0.34</td>
<td></td>
<td></td>
<td>pgf90_kdfloor</td>
</tr>
<tr>
<td>1.55</td>
<td>16.08</td>
<td>0.27</td>
<td></td>
<td></td>
<td>f90io_close</td>
</tr>
<tr>
<td>1.31</td>
<td>16.31</td>
<td>0.23</td>
<td></td>
<td></td>
<td>pgf90_sel_int_kind</td>
</tr>
<tr>
<td>1.12</td>
<td>16.50</td>
<td>0.19</td>
<td></td>
<td></td>
<td>__rouinit</td>
</tr>
<tr>
<td>1.01</td>
<td>16.67</td>
<td>0.17</td>
<td></td>
<td></td>
<td>annuityev_maxdate</td>
</tr>
</tbody>
</table>

Extract 4.1: Profile after manual optimisation of the interpolation routine

Comparing Extract 4.1 with Extract 3.1 shows that, by optimising the interpolation, we have reduced the proportion of time spent in that routine from roughly 40% to roughly 7%. This is a significant reduction, and is a direct result of the changes made in Chapter 3.

If we again remove __mcount2 from the analysis and combine the remaining routines so that those shown are consistent with Figure 2.2, then this profile may be illustrated as
It is clear from Figure 4.1 that the routine `annResSL`, which performs the reserving calculation for single-life annuity policies, now takes the largest proportion of the runtime, and so that routine should be investigated with a view to its optimisation.

It is also apparent that the number of functions appearing in Extracts 2.3, 3.1 and 4.1 increases as we progress with the optimisation. However, this is purely because the runtime for the entire program shortens at each stage, and so the proportion of time spent in the smaller routines has increased; the increase in the number of functions which are each more than 1% of the runtime is not a result of introduction of more routines into the program.

It is worth noting that the proportion of time spent in the `log()`, `exp()` and `power()` functions has increased as a result of the time spent in these functions remaining approximately constant while the overall runtime has reduced significantly. So far, we have not done anything to change how they are used; in Section 4.5 we discuss the removal of some calls to these functions.

### 4.2) Use of the Reserve Calculation

A *reserve* can be interpreted as ‘the current value of net assets which the assurance company needs to set aside in order to meet future liabilities’ [2]. To estimate the amount of the reserve, it is necessary to calculate the expected present value of the liability. For the policies under consideration in this project, we need to estimate the expected value of the payments to the pensioners.

The present value of the future payments to the policyholder can be calculated using methods which are elementary to actuarial science [2]. The basic concept is to allow for interest accruing on the current fund, and for the probability of making each payment. A slight complication to the calculation is introduced by allowing for escalation of the payments.
The actual derivation of the formulae required to calculate the reserves is considered in Appendix B. We find that the payment reserve factors, per £1 of benefit payable to the policyholder, are

\[ a_x = \frac{v^f}{l_x} \sum_{t=0}^{\infty} \left( 1 + e \right)^{\frac{f^t + f^t/m}{e^t}} v^t \left( \frac{f + \frac{f^t + f^t/m}{e^t}}{e^t} \right) \]  

(4.1)

and

\[ b_x = \frac{u^f}{l_x} \sum_{t=0}^{\infty} \left( 1 + e \right)^{\frac{f^t + f^t/m}{e^t}} u^t \left( \frac{f + \frac{f^t + f^t/m}{e^t}}{e^t} \right) \]  

(4.2)

and that, the renewal expense reserve factors, per £1 of renewal expense attributed to the policy, are

\[ a_x^{RE} = \frac{v^f}{l_x} \sum_{t=0}^{\infty} \left( 1 + e \right)^{\frac{f^t + f^t/m + k^t}{e^t}} v^t \left( \frac{f + \frac{f^t + f^t/m + k^t}{e^t}}{e^t} \right) \]  

(4.3)

and

\[ b_x^{RE} = \frac{u^f}{l_x} \sum_{t=0}^{\infty} \left( 1 + e \right)^{\frac{f^t + f^t/m + k^t}{e^t}} u^t \left( \frac{f + \frac{f^t + f^t/m + k^t}{e^t}}{e^t} \right) \]  

(4.4)

Hence, the total amount required to meet future payments with an initial amount of £P p.a. to a person now aged x, and associated renewal expenses with an initial amount of £R p.a., and investment expenses at a rate e p.a. is the total reserve \( V_x \), which is obtained as

\[ V_x = Pa_x + Ra_x^{RE} + \left[ P[b_x - a_x] + Rb_x^{RE} - a_x \right] \]  

(4.5)

Clearly, Equation (4.5) is algebraically identical to

\[ V_x = Ph_x + Rb_x^{RE} \]  

(4.6)

and this simplified version would lead to fewer calculations. However, each of the individual reserves are required separately for management reporting purposes, and therefore all of these calculations do need to be performed.

So far, we have justified the use of four reserve factors; those in (4.1) to (4.2). However, the commercial environment which inspired this investigation is a company which has a foreign parent company. Therefore, in addition to the need to satisfy UK regulations in order to trade in the UK, the company also needs to calculate reserves for payments and renewal expenses at an interest rate commensurate with the rate obtainable in the parent company’s country.

This means that, in the commercial environment, we need to calculate six reserves for each policy at each step of the projection. These reserves may be listed as;

1) payments, at the UK interest rate
2) renewal expenses, at the UK interest rate
3) payments, at adjusted UK interest rate to allow calculation of the investment expense reserve
4) renewal expenses, at adjusted UK interest rate to allow calculation of the investment expense reserve
5) payments, at the foreign interest rate
6) renewal expenses, at the foreign interest rate

In order to accurately emulate that process, the program under investigation in this report also calculates all six reserves at each step. Therefore, at each step we require six summations of the form

\[ a_x = \sum_{t=0}^{\infty} \left( 1 + e \right)^{\frac{f^t + f^t/m + k^t}{e^t}} v^t \left( \frac{f + \frac{f^t + f^t/m + k^t}{e^t}}{e^t} \right) \]  

where \( v = \frac{1}{(1+i)} \)  

(4.7)

An inspection of the code which performs this calculation suggested that there are several changes which could be made in order to enhance the performance of this routine.
4.3) Optimising the Reserve Calculation

In Section 1.4, we introduced the fact that this investigation uses some synthetic data which has the same shape as real data from the business. Since the real data comes in separate files, one for each different type of policy, the policies in the synthetic data under investigation are all the same type, i.e. single-life policies; in fact, for this investigation we only generated data for single life policies. So, for these data, we expect the program to only spend significant amounts of time in the function \texttt{annResSL}, which is precisely the function which calculates the annuity reserve for a single life. Such expectation is clearly consistent with the profile in Extract 4.1 and Figure 4.1.

4.3.1) Using Features of the F90 Language

Although experience from optimising the interpolation routine, in Chapter 3, suggests that updating code does not have a significant effect on runtime, updating code to take advantage of language features was still considered worthwhile since it aids maintainability of the code. Therefore, to mirror that Chapter, the first step in changing this method was to make the function \texttt{pure}, and add the required \texttt{intent()} clauses, and the second step was to change the declaration of all double precision variables to \texttt{real(8)}.

4.3.2) Removing Array Initialisations

Having made the changes described in Section 4.3.1, the first 18 lines of the function are:

\begin{verbatim}
pure function annResSL(actualAge1, mortality1, valnMonth, intRate, piMonth, escRate, escMonth, paymentsPerYr, delayInMonth, gteeRemainEnd) result(annuityReserve)
    real(8), intent(in) :: actualAge1, intRate, escRate, delayInMonth, gteeRemainEnd
    type(mortalityTable), intent(in) :: mortality1
    integer(2), intent(in) :: valnMonth, piMonth, escMonth, paymentsPerYr
    real(8) :: annuityReserve
    integer(2) :: intervalMonths, firstMonth, paytMonth(0:1500)
    real(8) :: vFactor, gtRemainEnd
    real(8) :: paytTime(0:1500)
    real(8) :: age(1:2,0:1500), lx(1:2,0:1500), prAlive(1:2,0:1500)
    real(8) :: paymentAmount, prPayt(0:1500), pvPayt
    real(8) :: escFac, escalationFactor(0:1500)
    integer k, maxK, m
    paytTime(0:1500) = 0.0d0
    age(1:2,0:1500) = 0.0d0
    lx(1:2,0:1500) = 0.0d0
    prAlive(1:2,0:1500) = 0.0d0
    prPayt(0:1500) = 0.0d0
    escalationFactor(0:1500) = 1.0d0

    Extract 4.2: The start of the annuity reserve {single life} function
\end{verbatim}

Initialising arrays takes time and so it seemed sensible to remove such initialisations where possible. However, it should be noted that not all annuity polices have payments which escalate, and in those cases just overwriting the existing values of the array \texttt{escalationFactor(:)} at the point the escalation is calculated serves no purpose, simply because no such overwriting is performed since no escalation calculations are invoked. Therefore totally removing the initialisation of \texttt{escalationFactor(:)} leads to incorrect program behaviour, and hence incorrect results.

In order to see whether the initialisations which do not relate to escalation are actually necessary, those arrays which are set to zero were instead set to \texttt{1000.0d0}. The fact that changing the initial values in these arrays does not alter the final results indicates that initialisation to zero is superfluous.

To allow for the lack of need for initialisation in some cases, the loop where the overall calculation is combined has been enclosed in an \texttt{if}-block which deals with whether or not escalation applies. For policies which have escalation, the initialisation of the escalation factor has been moved into this \texttt{if}-block; for policies without escalation, such initialisation has been completely removed.
The newly created block also allows the removal of some multiplication operations for cases with no escalation. This simplification of the summation has been further enhanced by allowing for the number of payments per year after the summation is complete, rather than accounting for it within each payment.

4.3.3) Shortening Arrays

From Extract 4.2, it is also apparent that all arrays are declared to allow for 1500 steps through time. However, from Section 1.4, we know that the input data shows that the maximum outstanding term for a policy is less than 65 years, or 780 months. We have therefore allocated shorter arrays to see whether this has any effect on time required to access memory or cache.

To allow for the very likely possibility that other data files will have different maximum outstanding terms, and hence the need to change the array lengths, the maximum array length has been defined as a pre-processor variable.

It is worth re-iterating that this function calculates the reserve for single-life annuities, and so no information relating to a second life is passed in to the function. There is therefore no need to declare arrays to hold such information and so, in a further attempt to reduce memory access and cache access times, relevant arrays have been changed to be one-dimensional.

4.3.4) Simplifying the Calculations

Several trivial simplifications now became apparent. The most obvious was the repeated division-by-12, which have all been replaced by multiply-by-ONE_TWELFTH, where ONE_TWELFTH is declared as a pre-processor variable.

Given the success of the permutations in Section 3.3.4, it seemed reasonable to attempt some investigation into whether such permutations have success here. The actual statements to be considered are

\[
\begin{align*}
\text{a} & \quad \text{intervalMonths} = \text{floor}(12.0d0/\text{paymentsPerYr} + 5.0D-4) \\
\text{b} & \quad \text{firstMonth} = \text{piMonth} - \text{valnMonth} \\
\text{c} & \quad \text{firstMonth} = \text{firstMonth} + \text{floor}(\text{dble}(\text{intervalMonths-firstMonth})/\text{intervalMonths}) * \text{intervalMonths} \\
\text{d} & \quad \text{age}(0) = \text{actualAge1} \\
\text{e} & \quad \text{lx}(0) = \text{lxInterp(actualAge1, mortality1%lx)} \\
\text{f} & \quad \text{gtRemainEnd} = \text{gteeRemainEnd-dble(1.0d0/\text{paymentsPerYr})+ 5.0D-4} \\
\text{g} & \quad \text{vFactor} = 1.0d0/(1.0d0+\text{intRate}) \\
\text{h} & \quad \text{annuityReserve} = 0.0d0
\end{align*}
\]

Extract 4.3: Initialisation statements considered for rearrangement

By inspection, it is clear that a and b must both precede c, but not necessarily in either particular order. This means that there are two possible permutations of these three lines. However, none of the other statements depend on any other, and so they may be placed in any order. Hence, the number of permutations may be calculated as:

for a, b, and c, there are 2 possibilities
for d may be placed in any of the 4 positions resulting from allocating a, b and c
for e may be placed in any of the 5 positions resulting from allocating a to d
for f may be placed in any of the 6 positions resulting from allocating a to e
for g may be placed in any of the 7 positions resulting from allocating a to f
for h may be placed in any of the 8 positions resulting from allocating a to g

\[ \Rightarrow \text{total number of permutations is } 2 \times 4 \times 5 \times 6 \times 7 \times 8 = \frac{8!}{3} = 13440 \]

Clearly, deriving all of these possible permutations, and then testing how long they take to run, is beyond the scope of a short investigation like this. Using the fact that the code currently runs in roughly 125 sec for the 1000 policies in the test dataset, a simple estimate for the overall runtime is

125 sec per run for each of 4 repetitions for each of 13440 permutations \( \approx \) 1867 hours
Even removing the repeated submissions, which check for stability of the runtime, and only running 50 of the 1000 policies gives an estimate of
\[
\left( \frac{125 \times 50}{1000} \right) \text{ sec per run for each of 13440 permutations } \approx 23 \text{ hours}
\]

However, a few attempts at trial-and-error leads to the discovery that the sequence \{a, f, e, g, b, d, h, c\} does run marginally faster than the sequence in Extract 4.3, although the difference in performance is certainly not as significant as the gain from rearranging the initialisation in the interpolation routine, as seen in Section 3.5.

Extract 4.3 indicates that several of these initialisations involve division by an integer. However, by the nature of the policies, for any particular policy that integer has a constant value, and so, allowing for the fact that division is a slow operation, it seemed reasonable to attempt to remove these divisions. A select-case block has therefore been introduced to allocate values, rather than perform calculations, where possible.

The next candidate for simplification was the population of the array which holds the probability that each future payment is made. At this stage of the investigation, the array is populated using by the statements;

```
    lx(0) = lxInterp(actualAge1, mortality1%lx)
    do while(k .le. maxK)
        lx(k) = lxInterp(age(k), mortality1%lx)
        prAlive(k) = lx(k) / lx(0)
        prPayt(k) = prAlive(k)
        k = k+1
    end do
```

Extract 4.4: Calculation of probabilities

The body of this loop contains a superfluous allocation and a division, neither of which can be benefiting performance. So, in a few distinct steps to allow for measuring timings, the loop was altered to produce

```
    reciprocal = 1.0d0 / lxInterp(actualAge1, mortality1%lx)
    do while(k .le. maxK)
        prPayt(k) = lxInterp(age(k), mortality1%lx) * reciprocal
        k = k+1
    end do
```

Extract 4.5: Simplified calculation of probabilities

It is clear, by comparing Extract 4.4 and Extract 4.5, that the allocation to \( prAlive(k) \) has been removed, and this is consistent with the assertion, just made, that there was a superfluous allocation.

### 4.3.5) Calculation of Discount Factors

The final summation in this function performs the amalgamation of the elements of the reserve factor. Using the representation introduced in Section 4.2 we have

\[
    a_x = \sum_{t=0}^{\infty} \left[ \frac{v^{t+1} / m^{t+1}}{v^{t+1} / m^{t+1} / m_t \cdot \frac{1 + e^{r_t}}{m_t / m}} \right]
\]

After having been initialised with \( \text{annuityReserve} = 0.0 \cdot d0 \), this is coded as

```
    do k = 1, maxK
        annuityReserve = annuityReserve + prPayt(k) * escalationFactor(k) * vFactor**paytTime(k)
    end do
```

Extract 4.6: Final summation of annuity factor

However, as we shall see in Section 4.5, calculating powers is slower than other forms of calculation, and so we decided to look for some other means of performing the calculations which provide the discount factors. It quickly became apparent that the \( v^t \)'s in this summation are in geometric progression, and so it was possible to introduce an array to hold the discount factors, and populate the array using the properties of the geometric progression. This will be discussed further in Section 4.5.
4.3.6) Speed of Light

Having reached this point, it was difficult to find areas of code which could be changed to produce significant performance gains, so we again used the "speed of light" approach to estimate the best possible runtime. To achieve this, the code which performs the calculations in the body of this routine was replaced by a nonsense version which just returns a hard-coded number.

4.4) The Effect on Performance

The actual runtimes for each stage of the optimisation are

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
<th>Runtime (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>04\a1</td>
<td>make annResSL pure &amp; only pass-in one mortality table</td>
<td>158.9</td>
</tr>
<tr>
<td>04\a2</td>
<td>change double precision to real(8)</td>
<td>157.2</td>
</tr>
<tr>
<td>04\b0</td>
<td>remove explicit initialisation :: check validity</td>
<td>158.7</td>
</tr>
<tr>
<td>04\b1</td>
<td>remove explicit initialisation :: paytMonth();</td>
<td>157.2</td>
</tr>
<tr>
<td>04\b2</td>
<td>remove explicit initialisation :: paytTime();</td>
<td>153.9</td>
</tr>
<tr>
<td>04\b3</td>
<td>remove explicit initialisation :: age(1:2,:)</td>
<td>145.0</td>
</tr>
<tr>
<td>04\b4</td>
<td>remove explicit initialisation :: lx(1:2,:)</td>
<td>137.4</td>
</tr>
<tr>
<td>04\b5</td>
<td>remove explicit initialisation :: prAlive(1:2,:)</td>
<td>130.8</td>
</tr>
<tr>
<td>04\b6</td>
<td>remove explicit initialisation :: prPayt();</td>
<td>127.9</td>
</tr>
<tr>
<td>04\b7</td>
<td>relocate explicit initialisation :: escalationFactor();</td>
<td>125.6</td>
</tr>
<tr>
<td>04\c1</td>
<td>reduce length of array :: paytMonth();</td>
<td>124.7</td>
</tr>
<tr>
<td>04\c2</td>
<td>reduce length of array :: paytTime();</td>
<td>125.0</td>
</tr>
<tr>
<td>04\c3</td>
<td>reduce length of array :: age(1:2,:)</td>
<td>125.3</td>
</tr>
<tr>
<td>04\c4</td>
<td>reduce length of array :: lx(1:2,:)</td>
<td>124.6</td>
</tr>
<tr>
<td>04\c5</td>
<td>reduce length of array :: prAlive(1:2,:)</td>
<td>125.2</td>
</tr>
<tr>
<td>04\c6</td>
<td>reduce length of array :: prPayt();</td>
<td>124.9</td>
</tr>
<tr>
<td>04\c7</td>
<td>reduce length of array :: escalationFactor();</td>
<td>125.5</td>
</tr>
<tr>
<td>04\d1</td>
<td>make array one-dimensional :: age(1:2,:)</td>
<td>125.3</td>
</tr>
<tr>
<td>04\d2</td>
<td>make array one-dimensional :: lx(1:2,:)</td>
<td>127.9</td>
</tr>
<tr>
<td>04\d3</td>
<td>make array one-dimensional :: prAlive(1:2,:)</td>
<td>124.7</td>
</tr>
<tr>
<td>04\e1</td>
<td>simplify calculation of return summation :: part 1</td>
<td>125.1</td>
</tr>
<tr>
<td>04\e2</td>
<td>simplify calculation of return summation :: part 2</td>
<td>122.7</td>
</tr>
<tr>
<td>04\f1</td>
<td>change /12.0 to *ONE_TWELFTH :: part 1</td>
<td>122.7</td>
</tr>
<tr>
<td>04\f2</td>
<td>change /12.0 to *ONE_TWELFTH :: part 2</td>
<td>122.7</td>
</tr>
<tr>
<td>04\g</td>
<td>tidy-up the code</td>
<td>122.7</td>
</tr>
<tr>
<td>04\h</td>
<td>reorder initialisations</td>
<td>122.6</td>
</tr>
<tr>
<td>04\j1</td>
<td>add select case (paymentsPerYr) :: part 1</td>
<td>119.4</td>
</tr>
<tr>
<td>04\j2</td>
<td>add select case (paymentsPerYr) :: part 2</td>
<td>122.1</td>
</tr>
<tr>
<td>04\j3</td>
<td>add select case (paymentsPerYr) :: part 3</td>
<td>121.2</td>
</tr>
<tr>
<td>04\k1</td>
<td>change how prPayt is calculated :: part 1</td>
<td>119.5</td>
</tr>
<tr>
<td>04\k2</td>
<td>change how prPayt is calculated :: part 2</td>
<td>118.3</td>
</tr>
<tr>
<td>04\k3</td>
<td>change how prPayt is calculated :: part 3</td>
<td>118.3</td>
</tr>
<tr>
<td>04\k4</td>
<td>change how prPayt is calculated :: part 4</td>
<td>114.0</td>
</tr>
<tr>
<td>04\k5</td>
<td>change how prPayt is calculated :: part 5</td>
<td>114.1</td>
</tr>
<tr>
<td>04\m</td>
<td>remove calculation of v**t for each payment</td>
<td>59.0</td>
</tr>
<tr>
<td>04\z</td>
<td>speed of light</td>
<td>0.8</td>
</tr>
</tbody>
</table>

Table 4.1: Time to process 1000 policies as optimisation of reserving calculation progressed

From the results in Table 4.1, we can make several observations

1) the difference in runtimes between the speed-of-light version of this routine and the original version indicate that, before optimisation, the time spent in the routine to calculate reserves was around 157 seconds. Similarly, the difference in runtimes between the speed-of-light version of this routine and the final version indicate that, after optimisation, the time spent in the routine was around 58 seconds. We have therefore reduced the time spent in this routine by \( \left( 1 - \frac{58}{157} \right) \approx 63\% \). This reduction is equivalent to
improving the performance of this method by a factor of $\frac{157}{58} \approx 2.7$, and although this factor is not as great as the corresponding improvement for the interpolation calculation, it is still very significant.

2) versions 4a1 and 4a2 indicate that changing the code to take advantage of F90 features has no real effect on the runtime, although, as with the interpolation routine, these changes had the effect of improving readability and maintainability of the code.

3) versions 4b0 to 4b7 indicate that completely removing the initialisation of some arrays, and reducing the number of elements initialised in other arrays, reduced the runtime from about 158.7 sec to 125.6 sec. This reduction is roughly 21% of the overall runtime, and shows that not initialising arrays unnecessarily is a quick and simple way of improving performance.

4) versions 4c and 4d indicate that reducing the lengths of the arrays does not have a significant effect on the runtimes. This is consistent with the fact that the arrays are still too long to fit into cache, and even if they did fit, they are likely to have been ejected by something-else as the calculations progressed.

5) versions 4k1 to 4k5 indicate that simplifying the calculation of the probability reduced the runtime from 121.2 sec to 114.1 sec. This is a reduction of roughly 5.9% of the overall runtime, and is consistent with the fact that divisions have been replaced by multiplication by the relevant reciprocal.

6) version 4m indicates that replacing the power calculations using prior evaluation and storing the results in an array reduced the runtime from about 114.1 sec to roughly 59.0 sec, a reduction of roughly 48%. This indicates that removing the power calculations was an extremely worthwhile stage of the optimisation of this routine.

7) version 4z shows that if the calculations in the reserving calculation could be performed ‘in no time’, then the amount of time required for the whole program to run is ‘negligible’. This indicates that, during the execution of this program, the vast majority of time is spent in this reserving routine and its daughters.

These results show that the time spent in this routine has reduced by about 63%, and this is made up of three main changes;

1) removing unnecessary initialisation of arrays
2) simplifying calculations which involve divisions
3) replacing power calculations with multiplications.

4.5) Replacing the Power Calculations

Part of the optimisation of this routine involved replacing power calculations by multiplications which were then stored in an array. That led to the time spent in the routine falling from about 114.1 sec to about 59.0 sec, a reduction of roughly half; this reduction was significant enough to warrant further investigation.

4.5.1) How the Power Calculations were Replaced

The function investigated in this Chapter is the implementation of the calculation of (4.7), the reserving factor derived in Section 4.2 as

$$a_x = \sum_{i=0}^{\infty} \left( 1 + v \right) \left( \frac{1}{m^n f/m} \right)^i \left( \frac{\sum m^i f/m}{I_x} \right) \quad \text{where} \quad v = \frac{1}{(1+i)}.$$
In the original form, \( v^{\frac{f}{m}} \) is calculated separately for every \( t \geq 0 \). It quickly becomes apparent that the \( v^r \) factors in this summation are in geometric progression; if we consider time steps \( t \) and \( t+1 \) then the ratio of the \( v^r \)'s is

\[
\frac{v^{\frac{f}{m}}}{v_{\frac{f}{m}+1}} = v^{\frac{1}{m}}.
\]

This means that once we know the first value in the sequence, \( v^{\frac{f}{m}} \), all subsequent values can be obtained via multiplication, rather than through power calculations. Therefore, these discount factors could have been calculated using a recurrence relation, viz;

\[
\begin{align*}
DF_0 &= v^{\frac{f}{m}} \\
DF_{t+1} &= DF_{t} \times v^{\frac{1}{m}} & t > 0
\end{align*}
\]

We introduced an array to hold these discount factors, and populated it using the recurrence relation for the geometric progression, i.e.

\[
\begin{align*}
discountFactor(0) &= v^{-\frac{f}{m}} \\
discountFactor(k) &= discountFactor(k-1) \times v^{\frac{1}{m}} & k \geq 1
\end{align*}
\]

Using this approach, the \( k \)th element of the array translates directly to the discount factor required at the time of the \( k \)th payment, and so the discount factor can be retrieved from the array, rather than being calculated using powers. The code required to implement this approach is

```plaintext
k = 0
paytMonth(k) = firstMonth - intervalMonths
discountFactor(k) = vFactor ** (ONE_TWELFTH * dble(paytMonth(k)-1) + delayInMonth)

vFactor = vFactor ** (ONE_TWELFTH * intervalMonths)
do while(age(k).le.119.0d0)
    .
discountFactor(k) = discountFactor(k-1) * vFactor
    end do
    .
do k = 1,maxK
    annuityReserve = annuityReserve + prPayt(k) * escalationFactor(k)* discountFactor(k)
end do
```

Extract 4.7: Calculation of discount factors using an array

This code is just as maintainable as the previous version, and this is clearly a bonus for code which executes so much more quickly than the version which used power calculations.

4.5.2) Validation of Number of Power Calculations Replaced

Having seen how the power calculations were replaced, we now look at whether the performance gain is consistent with having replaced the power calculations, or whether something else may also have contributed to the gain.

It is straightforward to monitor how many times the calculation is performed, simply by tracking how many iterations of the loop we pass through. By declaring `numPowers` as a globally accessible `integer(8)`, with a relevant initialisation to zero, it is possible to output the number of calculations by insertion of increment statements in the code. By doing so, we find that, for a run which processed 1000 policies, `numPowers = 1194.9 \times 10^6`.
It is relatively simple to show that this counted value is certainly realistic for the problem under consideration, although actually **proving** the correctness of the value is far more difficult, and is probably outside the scope of our short investigation.

As discussed in Section 1.4, the synthetic data was produced so that it was a representation, at 31/12/2008, of some policies relating to people who recently retired, with their ages in the range from roughly 55 to roughly 70. However, there is nothing special about 31/12/2008; the calculations could be performed at any month-end, and so the logic below could be translated slightly to allow for the difference in dates.

The standard mortality tables, as used within the life assurance industry, provide rates up to age 120. Therefore, to allow for the fact that interpolation is carried out using two integer ages either side of the required fractional age, the calculations in this program must stop at age 119, otherwise we ‘fall off’ the end of the mortality table, i.e. we cannot obtain the higher values required in the interpolation.

The data used by the program have outstanding terms which are distributed randomly. We require a representative policy for this reasonableness check, otherwise our estimate could be skewed by randomness. In order to select a sensible representative policy, we take the one with the median outstanding term; an analysis of the data showed that median outstanding term to be 683 months.

The number of power calculations replaced is estimated in Appendix C, and the value obtained is $1190.8 \times 10^6$, and this estimate clearly agrees with the observed value of $1194.9 \times 10^6$ obtained by the insertion of a counter.

We then produced a small program to time how long iFort took to perform several multiplications and the corresponding number of power calculations. In order to ensure that the compiler does not treat all calculations as dead code, and therefore eliminate it, we need to print some statistic relating to each set of calculations; for simplicity we chose the sum of the elements. The body of the code for this is

```fortran
integer, parameter :: numElems = 5000000
real(8), dimension(1:numElems) :: a,b,c,d

integer k=1,numElems
a(k) = rand(0) * 1.0d2
b(k) = rand(0) * 1.0d2

end do

c(:) = a(:) * b(:)
d(:) = a(:) ** b(:)
write(\(2es11.3\) ) sum(c(:)), sum(d(:))
```

Extract 4.8: Timing power and multiplication calculations

The times taken were 0.027 sec for the multiplications, and 0.243 sec for the power calculations. Using these times, we have:

\[
\text{time saving for } 5 \times 10^6 \text{ calculations } = 0.243 - 0.027 = 0.216 \text{ sec}
\]

\[\Rightarrow \text{average saving for each calculation } = \frac{0.216}{5 \times 10^6} \text{ sec}\]

\[\Rightarrow \text{expected saving for number of calculations in program } = \left(\frac{0.216}{5 \times 10^6}\right) \times 1194.9 \times 10^6 \approx 51.6 \text{ sec.}\]

The observed time saving of roughly $114 - 59 = 55$ sec is clearly consistent with our estimated expected value of about 52 sec, and so we conclude that it is unlikely that factors other than the replacement of the power calculations have significantly affected the performance gain.
4.6) Other Types of Policy

This Chapter has discussed the optimisation of the subroutine \texttt{annResSL} because that is the routine used by the single-life policies in this synthetic data. In the commercial environment, the program emulated by the one in this investigation is used to value many different types of annuity policy. Each of these different types requires a slightly different reserving calculation, and these calculations are in routines which may be labelled \texttt{annResXX}, where \texttt{XX} could be

1) \texttt{RA} for Reversionary annuities; the payments are made to another person, generally the spouse, after the death of the policyholder.

2) \texttt{JL} for Joint-Life annuities; the payments are made so long as all the people specified in the policy remain alive.

3) \texttt{LS} for Last-Survivor annuities; the payments are made so long as at least one of the people specified in the policy remain alive.

Within each of these routines, the fundamental algorithm is the same; it is only the probabilities which differ significantly. Given the success of the optimisations in the function to calculate the single-life reserves, all the other reserving routines could be updated with corresponding changes.

These other routines are not utilised by the program when it processes this synthetic data, and hence optimising these routines can have no scientific benefit to this investigation. However, this investigation is founded in a commercial environment, and such changes will be beneficial in that setting. In line with good software engineering practice, we have updated the other reserving routines on the basis that the changes are likely to be beneficial in the commercial environment.

4.7) Summary

In this Chapter we have investigated the manual optimisation of the reserving calculation. We have performed many small changes and have discovered that those changes which had a major effect can be grouped into three categories;

1) removing unnecessary initialisation of arrays

2) simplifying calculations which involve divisions

3) replacing power calculations with successive multiplications.

In Section 3.3.1, we highlighted the change to using \texttt{real(8)} and suggested that such a change would make a natural basis for a further investigation. Clearly, we made equivalent changes in Section 4.3.1 and so we could extend the investigation, proposed in Section 3.6, into the effect of reduced precision on runtime to include the contribution from each of the routines optimised.

In Section 4.6, we updated several other routines with changes equivalent to those introduced for the single life reserve calculation. Discovering whether the effect of the changes was the same for each routine, or whether the performance gains were greater in some routines than in others, would also make an interesting basis for a subsequent investigation.
5) Portability of the Serial Optimisation

So far, this investigation has been performed on a dual-core laptop running Windows XP, because that provides a reasonable emulation of the processing power of the PCs currently available in the commercial environment. In this Chapter, we consider the effect of the serial optimisation on other platforms.

First, we re-profile the code as it is after the serial optimisation, and we discuss the progression of the profiles seen so far.

Next, we summarise the platforms used in this investigation. That short Section also includes an overview of the rationale for the inclusion of those platforms.

We then discuss the effect of the various stages of optimisation on the different platforms, and consider a summary comparison of the overall effects on the different platforms.

5.1) Re-profiling the Code

Until now we have performed several stages of optimisation on a serial program, and we have seen a significant improvement in performance. By re-profiling the code, we can see what scope there is for further significant performance gains.

5.1.1) Current Profile

The following Extract shows the profile of the updated code and, to maintain consistency with previous Extracts from profiles, only shows those routines which constitute more than 1% of the time.

<table>
<thead>
<tr>
<th>%</th>
<th>cumulative</th>
<th>time</th>
<th>seconds</th>
<th>self</th>
<th>seconds</th>
<th>total</th>
<th>self</th>
<th>calls</th>
<th>s/call</th>
<th>s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.83</td>
<td>2.33</td>
<td>2.33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>_mcount2</td>
</tr>
<tr>
<td>22.14</td>
<td>3.85</td>
<td>1.52</td>
<td>115859788</td>
<td>0.00</td>
<td>0.00</td>
<td>annuityev_lxinterp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.10</td>
<td>4.82</td>
<td>0.97</td>
<td>408108</td>
<td>0.00</td>
<td>0.00</td>
<td>annuityev_annressl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.18</td>
<td>5.31</td>
<td>0.49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>__rouexit</td>
</tr>
<tr>
<td>4.14</td>
<td>5.60</td>
<td>0.28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f90io_close</td>
</tr>
<tr>
<td>2.29</td>
<td>5.76</td>
<td>0.16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>annuityev_annresgt</td>
</tr>
<tr>
<td>2.23</td>
<td>5.91</td>
<td>0.15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pgf90_floor</td>
</tr>
<tr>
<td>2.23</td>
<td>6.06</td>
<td>0.15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pgf90_kfloor</td>
</tr>
<tr>
<td>2.02</td>
<td>6.20</td>
<td>0.14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pgf90_dfloor</td>
</tr>
<tr>
<td>1.91</td>
<td>6.33</td>
<td>0.13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pgf90_kdfloor</td>
</tr>
<tr>
<td>1.91</td>
<td>6.47</td>
<td>0.13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pgf90_sel_int_kind</td>
</tr>
<tr>
<td>1.80</td>
<td>6.59</td>
<td>0.12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>__rouinit</td>
</tr>
<tr>
<td>1.13</td>
<td>6.67</td>
<td>0.08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>annuityev_maxdate</td>
</tr>
</tbody>
</table>

Extract 5.1: Profile after manual optimisation of reserve calculation

Comparing Extract 5.1 to Extract 4.1 shows that, by changing the way the discount factors are calculated, the execution of the program is no longer dominated by routines __fvdlog_long and __fvdexp_long. This is clearly consistent with the reduction in the number of times the program performs power calculations, and the consequent need to calculate so many logs and anti-logs.

Extract 5.1 also shows that if we disregard _mcount2, on the basis that it will not appear in the ‘live’ program, then the majority of the time is spent in the interpolation and reserving routines. Since those routines have already been optimised, we have reached the position that the two routines which contribute most significantly to the runtime have been optimised in this investigation, and the remaining routines, which contribute slightly less significantly, are provided by the compiler writers and are therefore outside the bounds of what can realistically be optimised in this investigation.

Following our standard approach of combining the routines so that those shown are consistent with Figure 2.2, this situation may be illustrated as
By comparing Figure 5.1 with Figure 4.1, we can see that replacing the power calculations in Section 4.3.5 has led to a vast reduction in the proportion of time spent in those routines, and this is clearly consistent with the reduction in the amount of time taken, as discussed in Section 4.5.

It is therefore clear that significant gains in performance can only be made either by altering the algorithms used, so that the dominant routines are no longer prevalent, or by adding parallelism. Time constraints suggest that the benefits from parallelism might outweigh making major changes to the algorithms used.

5.1.2) Progression of Profiles

So far we have profiled the code several times, and there has been a discussion of the individual profiles near the beginning of each of the relevant Chapters. Each of those profiles has been compared to the previous profile, and those comparisons have been central to the order in which this investigation has progressed. However, because we concluded in Section 5.1.1 that there are no other routines which present themselves as candidates for optimisation within this project, it is sensible to consider the effect each phase of optimisation has had on the profile of the program.

By combining the information from each of the profiles, and excluding _mcount2, the progression of the time spent in each may be seen as
Figure 5.2 highlights many of the features discussed earlier in this report, viz;

1) the bar labelled ‘initial code’ shows the same features as Figure 2.2, i.e. just over half the time is spent in the reserving calculation, and roughly one third of the time is spent in the interpolation routine.

2) the bar labelled ‘compiler flags’ is consistent with Figure 3.1 and the discussion in Section 3.1 where it was noted that, as a result of tuning the compiler flags, the interpolation routine became more dominant than the reserving calculation.

3) the bar labelled ‘interpolation’ is consistent with Figure 4.1 and the discussion in Section 4.1 where we noted that, following the optimisation of the interpolation routine, the reserving calculation was the only remaining dominant routine.

4) the bar labelled ‘reserve’ is consistent with Figure 5.1 and the discussion in Section 5.1.1 where we noted that, following the optimisation of the reserving calculation, there are no other routines which are obvious candidates for serial optimisation.

These observations are consistent with the suggestion, in Section 5.1.1, that not much more increase in performance can be extracted from further serial optimisation, and hence the code should be parallelised in order to reduce runtimes further.

5.2) Platforms Used

In Section 1.5 we described the platforms used in this investigation. In summary, they are:

1) a laptop with a 2.0GHz dual-core CPU, running Windows XP, and using iFort
2) a desktop with a 2.6GHz quad-core CPU, running Windows XP, and using iFort
3) Ness, EPCC’s training machine, which can utilise up to sixteen 2.6GHz cores using Linux and PGF90.

The laptop is used to imitate the performance on the PCs available in the business, and has been the only platform used so far, with the obvious result that all discussions to date relate to the laptop. Since the laptop has a dual-core CPU, it will be used as the initial platform for the parallelisation stage of this investigation.

The desktop’s purpose is to see how well the code scales on a Windows XP platform, and this is a direct result of the machines in the business environment running Windows XP. The reason for including Ness is to see
whether the code scales well to the slightly higher number of cores which could be available in the business within the near future.

Ideally, in order to be consistent with machines which are likely to become available to the business in the near future, we would have used one PC with 16 cores, running Windows XP, for the whole investigation but such a machine was not available to this investigation.

This investigation is not primarily interested in comparisons between these platforms, and so the fact that such inter-platform comparisons are still slightly contaminated by different compilers, and operating systems, is not a major concern. However, it is still illuminating to consider an overview, on these other platforms, of the performance increases resulting from changes made so far.

5.3) Comparability of Serial Optimisation

Intel’s iFort and Portland’s PGF90 provide slightly different enhancements to the F90 standard. The code investigated in this project is written using F90 which is close to the standard, and so, with only slight modifications to allow for the compiler’s dialects, we could have run all versions of the code, on the desktop and on Ness, to see the progression of performance gains resulting from each of the changes to the code. However, at this stage of the investigation, we are not interested in performing a complete re-analysis of the serial optimisation, and so we only run selected versions of the code which highlight certain areas of interest.

5.3.1) Compiler Options

In Section 2.3.2 we investigated the effect of various compiler options on the code’s performance on the laptop. We concluded that, on the laptop, the default level of optimisation was /O2 and that using /fast was the most beneficial option.

Whilst we note that the PGF90 flags are actually -O3 and -fast, in the following discussion we use /O3 and /fast purely for consistency with iFort; this is solely to avoid changing between the two sets of terminology, and is of no consequence to the conclusions drawn. Further, the man pages for PGF90 suggest that it is not possible to completely disable optimisation; either /O1 or /O2 is used according to whether or not /g is specified.

For the comparison between platforms, we might assume that /fast is the most beneficial option, although we needed to verify this, and so performed a summary investigation. The runtimes were

<table>
<thead>
<tr>
<th>Compiler Option Version</th>
<th>Code Version</th>
<th>Runtime (sec)</th>
<th>Desktop</th>
<th>Ness</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>00</td>
<td>232.3</td>
<td>544.6</td>
<td></td>
</tr>
<tr>
<td>disabled</td>
<td>01\m0d</td>
<td>589.3</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>O1</td>
<td>01\m01</td>
<td>323.1</td>
<td>543.4</td>
<td></td>
</tr>
<tr>
<td>O2</td>
<td>01\m02</td>
<td>232.3</td>
<td>325.7</td>
<td></td>
</tr>
<tr>
<td>O3</td>
<td>01\m03</td>
<td>232.3</td>
<td>325.6</td>
<td></td>
</tr>
<tr>
<td>fast</td>
<td>01\h</td>
<td>188.9</td>
<td>234.6</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Time to process 1000 policies using various compiler options

From the results in Table 5.1, we can make several observations;

1) the default level of optimisation has the same runtime as /O3 on the desktop, and /O1 on Ness. In Section 2.3.2 we deduced that the default level of optimisation on the laptop is /O2. In terms of this investigation, the observation that the default level of optimisation differs across platforms is only mildly interesting. However, for the business environment, this is a significant discovery; it implies that the compiler installation on the office PCs has a less than optimal level of optimisation, and this obviously has a detrimental effect on all programs produced in that environment. Clearly the business installation should be changed so that /O3 is the default.

2) on all three platforms, /fast produces faster code than /O3. We know, from Section 2.3.3, that /QxT (inclusion of SSE instructions) is the component of /fast which has the main additional benefit to /O3 on
the laptop. However, as just stated, we are not performing a full re-analysis of performance, and have therefore not investigated whether each of the components of /fast has the same effect on all platforms.

3) the laptop and desktop both use iFort, so we expect the code produced with a particular optimisation level to be consistent. When the code is compiled using /fast, the laptop takes roughly 267 sec and the desktop takes roughly 189 sec. Converting these times into a rate of processing we obtain

- laptop: \( \frac{1000}{267} \approx 3.7 \) polices per second
- desktop: \( \frac{1000}{189} \approx 5.3 \) polices per second

The ratio of these rates is \( \frac{5.3}{3.7} \approx 1.4 \), so that the desktop is processing about 40% more policies per second than the laptop. The core’s clock frequencies are 2.0GHz for the laptop and 2.6GHz for the desktop, and so the ratio of the frequencies is \( \frac{2.6}{2.0} = 1.3 \). Comparing these two ratios suggests that the rate of processing is proportional to the core’s frequency. Although there is a slight difference in the ratios, that difference may be due to considerations such as the desktop having more cores which may be utilised by the operating system, thereby leading to less switching out of the program being timed, so that other daemons may be dealt with.

Since Ness uses PGF90, we cannot argue that the executable produced should be identical to those produced by iFort. However, the fact that Ness takes roughly 235 sec when /fast is used is encouraging; we can obtain the equivalent comparison. The rate of processing becomes

- Ness: \( \frac{1000}{235} \approx 4.3 \) polices per second

The ratio of this with the laptop’s equivalent gives \( \frac{4.3}{3.7} \approx 1.2 \), so that Ness is processing about 20% more policies per second than the laptop. The frequency of the cores in Ness are 2.6GHz, and so the ratio with the laptop’s frequency is \( \frac{2.6}{2.0} = 1.3 \). Again, the closeness of these ratios suggests that the rate of processing on Ness is proportional to the core’s frequency. The figures given here contain rounding errors which may hide some difference due to considerations such as Ness using linux and PGF90, rather than Windows and iFort, but the similarity of the ratios suggest that these differences must be small.

These results suggest that, irrespective of a platform’s default options, when /fast is used the performance of the code is roughly proportional to the core’s frequency.

### 5.3.2) Optimisation of the Interpolation Routine

In Chapter 3 we performed manual optimisation of the interpolation routine on the laptop, and that had the effect of significantly reducing the overall runtime. We now look at the effect the optimisation of that routine had on the other platforms. To be consistent with Section 5.3.1, we only performed a summary investigation. The runtimes were

<table>
<thead>
<tr>
<th>Description</th>
<th>Code Version</th>
<th>Laptop</th>
<th>Desktop</th>
<th>Ness</th>
</tr>
</thead>
<tbody>
<tr>
<td>before manual optimisation</td>
<td>02'd</td>
<td>266.8</td>
<td>186.4</td>
<td>229.7</td>
</tr>
<tr>
<td>after loop unrolling</td>
<td>03'h3</td>
<td>228.5</td>
<td>162.9</td>
<td>205.9</td>
</tr>
<tr>
<td>simplification of arithmetic</td>
<td>03'c8</td>
<td>181.8</td>
<td>92.8</td>
<td>157.1</td>
</tr>
<tr>
<td>reordering statements</td>
<td>03'f06</td>
<td>158.0</td>
<td>92.1</td>
<td>155.0</td>
</tr>
<tr>
<td>speed-of-light</td>
<td>03'z</td>
<td>155.2</td>
<td>90.0</td>
<td>123.8</td>
</tr>
</tbody>
</table>

Table 5.2: Time to process 1000 policies during manual optimisation of the interpolation routine

From the results in Table 5.2, we can make several observations;

1) on the desktop and Ness, the most significant increase in performance was from simplifying the arithmetic, while the reordering of statements has almost no effect. This differs from the laptop where reordering the statements certainly had a measurable effect, and confirms that the significance of reordering may not be as great as suggested in Chapter 3.
2) the speed-of-light runtime can be used to estimate the speedup, achieved through manual optimisation, for this routine. Using \( \text{speedup} = \frac{\text{original} - \text{speed of light}}{\text{final} - \text{speed of light}} \) we have

\[
\begin{align*}
\text{laptop: speedup} &= \frac{266.766 - 155.186}{157.953 - 155.186} \approx 40 \\
\text{desktop: speedup} &= \frac{186.359 - 90.049}{92.141 - 90.049} \approx 46 \\
\text{Ness: speedup} &= \frac{229.710 - 123.676}{154.962 - 123.676} \approx 3.4
\end{align*}
\]

These speedups clearly indicate that the improvement on the laptop and desktop are far larger than that for Ness.

3) on the desktop and the laptop, the speed-of-light version executes in only a slightly shorter time than the final version, whereas on Ness the speed-of-light version executes in a significantly shorter time than the final version. This suggests that further optimisation on Ness may be possible.

These results indicate that manual optimisation of this routine was clearly beneficial on all platforms. While there has been significant improvement in performance on all platforms, the main source of the performance gain, and the extent of the benefit, differed between the platforms.

5.3.3) Optimisation of the Reserving Calculation

In Chapter 4 we manually optimised the reserving calculation on the laptop and, again, that significantly reduced the overall runtime. We now consider at the effect that optimisation had on the other platforms. To be consistent with the two previous subsections, we only performed a summary investigation. The runtimes were

<table>
<thead>
<tr>
<th>Description</th>
<th>Code Version</th>
<th>Laptop (sec)</th>
<th>Desktop (sec)</th>
<th>Ness (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>before manual optimisation</td>
<td>03:f06</td>
<td>158.0</td>
<td>92.1</td>
<td>155.0</td>
</tr>
<tr>
<td>removal of initialisation of arrays</td>
<td>04:b7</td>
<td>125.6</td>
<td>73.3</td>
<td>113.9</td>
</tr>
<tr>
<td>miscellaneous simplifications</td>
<td>04:k5</td>
<td>114.1</td>
<td>69.2</td>
<td>105.3</td>
</tr>
<tr>
<td>replacing power calculations</td>
<td>04:m</td>
<td>59.0</td>
<td>38.9</td>
<td>51.4</td>
</tr>
<tr>
<td>speed-of-light</td>
<td>04:z</td>
<td>0.7</td>
<td>0.6</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 5.3: Time to process 1000 policies during manual optimisation of the reserving calculation

From the results in Table 5.3, we can make several observations;

1) on all three platforms, the speed-of-light version executes in a particularly short time, indicating that the majority of the runtime for the whole program is spent in the reserving calculation and its daughter routines.

2) removing the initialisation of the arrays reduces the overall runtime by about 20% on the laptop and the desktop, which both use iFort, while the reduction is about 25% for Ness which uses PGF90. These are reasonably large gains, and they are effectively ‘free’ because no alternative calculations need to be performed in order to ensure that the correct result is produced.

3) on all three platforms, the various simplifications further reduced the time spent in this routine. The reduction is between 5% and 10% depending on the platform, and although this is not as great as the previous gain, it is still significant.

4) on all three platforms, replacing the power calculations by multiplications further reduced the time spent in this routine by roughly 50%.

5) using the speed-of-light runtime to estimate the speedup, achieved through manual optimisation, gives

\[
\begin{align*}
\text{laptop: speedup} &= \frac{157.953 - 0.735}{58.952 - 0.735} \approx 2.7 \\
\text{desktop: speedup} &= \frac{92.141 - 0.610}{38.876 - 0.610} \approx 2.4
\end{align*}
\]
Ness: speedup \( \approx \frac{154.962 - 0.979}{51.382 - 0.979} \approx 3.1 \)

These results indicate that manual optimisation of the reserving calculation was beneficial, although the magnitude of the improvement for this routine was not as great as for the interpolation routine. While the main source of the benefit from tuning the interpolation routine differed between the platforms, the main benefits from tuning the reserving calculation were the same on all platforms, i.e. replacing the power calculations and removing the unnecessary initialisation of the arrays.

### 5.3.4) Comparison of Improvement in Performance

In the preceding few Sections, we have seen how each phase of the optimisation has separately improved performance. It is illuminating to consider the overall improvement as each of the phases is applied, and this can be seen as

![Cumulative Speedup during Serial Optimisation](image)

Figure 5.3: Cumulative speedup during serial optimisation

Figure 5.3 highlights many of the features discussed earlier in this Section, viz:

1) the laptop gains more benefit from the application of /fast than the other two platforms; this is clearly a result of the fact that the laptop has a lower level of default optimisation that the other platforms.

2) the manual optimisation of the two routines each had a clear, and cumulative, effect on the performance of the code.

The shape of the cluster for all phases is similar, and this suggests that, once /fast is invoked, the effect of each phase is similar on all platforms. This can be seen more easily by considering the speedup of the individual phases, viz;
Figure 5.4: Speedup resulting from individual phases of serial optimisation

Figure 5.4 highlights the fact that the manual optimisation of the two routines had a similar effect on performance on each of the platforms.

These observations suggest that the serial optimisation performed so far is independent of the platform on which the code runs. This independence vindicates the use of the profile from Ness to drive the optimisation phases on the laptop.

5.4) Summary

In this Chapter we have considered the serial optimisation of our code as it applies to other platforms. We have seen that although the actual benefit from any particular phase differs across platforms, the overall effect was to significantly reduce the runtime on each platform.

The difference in speedup on the different platforms may be of some interest for a future investigation. Whether the difference is a real feature of the compilers, or whether there are many confounding factors, such as different operating systems and daemons, could be investigated using both iFort and PGF90 on a clean installation of Windows XP.

We have seen a significant improvement in performance for all platforms. From Sections 2.3.2 and 5.3.1 we know that moving to /fast was one of the factors which was beneficial on each platform. Section 2.3.3 showed that the inclusion of SSE instructions via /QxT was one of the factors of /fast which contributed to that improvement on the laptop. It therefore seems reasonable to infer that SSE instructions have some benefit on all platforms. A direct consequence of this is that any future work, resulting from this investigation, should try to find more opportunities for vectorisation within the code, and hence increase the benefit from /QxT.
6) Implementing and Testing Parallelism

So far, this investigation has considered the serial optimisation of the code, and in Section 5.1.1 we concluded that the serial phase could be brought to a close since there were no other routines which lent themselves to manual optimisation. Consequently, the remainder of this report considers the parallelisation of the code. This Chapter considers the addition, and initial testing, of parallelism, and Chapter 7 discusses the scalability of the parallel code.

First, we consider the rationale behind our choice of parallelisation, and the position which our strategy takes within the business environment. Our chosen approach leads to the need to re-engineer the code, and so we next discuss whether there have been any improvements in performance as a result of this re-engineering.

Next, we discuss the parallel scaling of the code as it processes the same 1000 policies we have used throughout the earlier Chapters of this report.

Finally, we consider the changes which are required to enable this code to process the numbers of polices which are in the datasets in the commercial environment.

6.1) The Approach to Parallelism

The code studied in this investigation was intended to imitate the code produced by the valuation package as closely as possible so that any conclusions drawn in the investigation may be translated back to the commercial environment if the relevant modifications were able to be made in the valuation package. To be consistent with this, any examination of parallelism must be founded on the ability to include an equivalent parallel implementation in the commercial environment; we therefore need to consider how parallelism fits in that environment.

6.1.1) Choice of Parallel Model

Starting from the business perspective, we considered two main choices of models of parallelism for use in this investigation; the message-passing model, implemented using MPI, and the shared-memory model, implemented using OpenMP. Both of these models have features which are attractive.

The fact that the code in this investigation imitates the code from the valuation package means that all variables, with the exception of those declared within routines optimised in Chapters 3 and 4, are declared so that their scope is global. Under the MPI approach, each core runs a thread which has access to a complete copy of the program, and that copy is independent of the copies visible to all other cores. Therefore, the global scope of variables does not present any problems to an MPI parallelisation. However, far more attention needs to be paid to variables’ scope under the OpenMP approach.

The program is written in a way that roughly 300 variables are required for the processing, with only 15 variables being read from the data, making the program small enough to fit into a relatively small amount of memory. There is therefore no requirement for the multi-gigabyte quantities of memory available to large, distributed-memory machines. Many shared-memory machines will have sufficient memory to hold multiple copies of the program, and the required data, making these machines equally favourable for this program.

For a program which requires data that will fit into memory, it is reasonable for the first evolution of an MPI version of that program to read the all data on a single thread, and then distribute whatever data is necessary. Therefore, for this investigation, the intricacies of parallel i/o in MPI are also not a relevant factor.

Having considered these factors, the final choice of model has been made for business, rather than technical, reasons. Firstly, an OpenMP library was already an integral part of the iFort installation, whereas an MPI library would have to have been integrated. Secondly, this investigation was born in a commercial environment where the majority of staff are not conversant with HPC, and so shared-memory PCs, with multi-core chips in the same box, are easily understood, whereas MPI could easily be misunderstood; there could be a suggestion that linking the 3500 PCs in the building into a single cluster would have the same processing abilities as a specially built supercomputer. Whilst building such a cluster would be technically possible, the effort required...
to do this would probably outweigh the benefit, not least because of the practical difficulty of coordinating a processing job using machines spread so widely throughout the company.

Other business considerations include maintainability of code, and cost of hardware. The code will inevitably need to be maintained, and an OpenMP code could be changed by programmers who are not HPC specialists without too many detrimental effects, whereas those same programmers may find it difficult to correctly modify a working MPI implementation. Hardware considerations come from the fact that the numbers of policies involved do not require sufficient parallel processing to warrant the investment in a large distributed-memory machine.

Speedup attainable is also a consideration. The code produced by the valuation package processes at a rate of roughly one policy per second, so that a dataset with 160,000 policies takes about 44 hours to process. In the business environment, a speedup of 100 is sufficient because that means that those 160,000 policies could be processed in less than half an hour. A speedup factor of this order may be achievable using OpenMP; the speedups which MPI may produce could be considered excessive because there is no commercial benefit in processing the policies in half a minute, rather than half an hour.

Therefore, although MPI is favoured in terms of programming simplicity and potential scaling, we have chosen the OpenMP approach, based primarily on the recognition that, in the business environment, far smaller numbers of cores are likely to be available, and OpenMP is simpler to maintain.

6.1.2) Choice of Processing Technique

Having chosen an OpenMP approach, there are several possibilities for actually parallelising the program. OpenMP is designed so that it is possible to parallelise only part of a program, and there are two parts of our program which are obvious candidates to be parallelised; the loops within the reserve calculation discussed in Chapter 4, and the loops representing the individual monthly timesteps of the projection. However, if we adopt an approach of parallelising only part of the program, then many processors would be idle during the serial portions of the program, and this obviously has detrimental consequences on the potential efficiency of the code.

The result of processing the data is a file containing information on each policy. For the purposes of the reserving and profitability calculations performed in this program, each policy is independent, and so we can perform a decomposition of the input data [12]. Further, since all policies will potentially have different outstanding terms, we regard this as unbalanced data decomposition, although, in general, the difference in the amount of processing required by each policy is likely to be reasonably modest.

By regarding the processing of each policy as a separate task, it is apparent that the number of tasks greatly exceeds the number of processors that are likely to be available on machines used to run this program. In these circumstances, it may be beneficial to include dynamic load balancing so that all processors are kept busy until all the tasks have been allocated. If such load balancing was not included, then some processors may finish far in advance of others and therefore be idle while other processors still have work to do. Clearly, the more accurate the balance, the sooner the overall job can finish.

We considered various methods of implementing this data decomposition model. One was to split the datafile, have each fragment processed by a separate thread, and then output the results to separate files which would need to be recombined later. However this has a drawback; it is impossible to know, without some analysis of the outstanding terms of the policies, whether the processing required for each file will be roughly equal. A more fundamental concern lies in the logistical problem of re-combining the results from these separate files into the format which the business reporting procedures use.

We finally decided on an implementation which uses a single thread to read large numbers of policies into a shared array, process all of the policies using a team of threads, store the results to another shared array, and finally write the results out to a single file using a single thread.
This approach is illuminated by the following snippet of pseudocode;

```pseudocode
open inFile for input as #1
open outFile for output as #2

do k = 1,1000
   input #1, policyDetails(k)
end do

parallel do
   do k = 1,1000
      policyResults(k) = process( policyDetails(k) )
   end do
end parallel do

do k = 1,1000
   write #2, policyResults(k)
end do

close #1
close #2
```

Extract 6.1: Pseudocode for implementation of data decomposition

This approach has the added advantage that policies appear in the results file in the same order they appear in the datafile. This is beneficial because testing simplifies to comparing the output file from the current evolution of the program to a file which is known to be correct. As the testing is simple, it is likely to be more reliable.

6.1.3) Re-engineering the Code

This investigation is based on an emulation of the F77 code produced by the valuation package. As part of the process of emulating that F77 code as closely as possible, the F90 used in this investigation was written so that the majority of the variables in the serial code have global scope. OpenMP requires all variables used within a parallel region to be declared as being either shared or private, unless the variables are declared in a routine called from within a parallel region, in which case the variables are private to the thread which is processing that region [13]. Therefore, in order to successfully implement OpenMP parallelism, the program should be re-engineered so that variables used within a parallel region no longer have global scope.

For those variables which hold the economic, temporal and demographic assumptions required to process the data, the solution was to make them read-only by declaring them as parameters using the specific feature of F90. However, the majority of variables had to be made more local to the routines to which they relate.

During the re-engineering, we took the opportunity to move away from the monolithic code layout which resulted from the imitation of the valuation package’s code. This restructuring was achieved mainly by the introduction of F90 modules which contained both user-defined-types and the methods which should be applied to them in order to emulate an object-oriented approach [14]. Since F90 is not a true object-oriented language [15], and this is an investigation into performance, rather than being a software engineering project, our re-engineering uses the spirit of object-orientation to gather related concepts into modules; we have not attempted to fully reproduce an object-oriented program.

In F90, it is possible to implement the OOAD philosophy of encapsulation by using the private modifier to enforce data-hiding within each module [16]. We have not declared any variables in this way, and this has the effect of making any variables declared at a module level accessible throughout the program. This circumvents the need to create interfaces into routines which modify these data; a clear advantage in this short investigation.

We also used the re-engineering to move all the global ‘read only’ variables into separate modules. This had the effect of placing all variables which may need to be changed in a readily accessible collection. We also took many of those variables which had been declared as pre-processor variables during the serial optimisation phase, and changed them to be F90 parameters in the modules which contained the other read-only parameters; this was in accordance with the suggestion that so-called “constant folding" is beneficial [7].

Having moved the relevant variables so that they are declared within subroutines and modules, the actual parallelisation using OpenMP was performed in one routine. This is a direct result of those moved variables
now being private to the thread on which they are declared. Therefore, performing such a substantial re-engineering exercise was not only beneficial from a software engineering perspective, with the updated code being easier to maintain, it also greatly simplified the process of adding OpenMP.

Throughout this re-engineering, the performance of the code was regularly monitored. Each of the runtimes, on the laptop, was in the range 57.0 sec to 58.2 sec, and this variation was not considered to be significant enough to warrant further investigation. The only slight point of interest is that one of the steps in this process was to alter how the data are read from the data file so that, rather than using one long line of code, we use several distinct lines, each having the ‘advance = no’ clause. This had the effect of increasing the overall time by a small but distinctly measurable amount, i.e. roughly 0.17%. However, that increase is still an insignificant proportion of the overall time, and since the time for actual processing time did not increase as a result of this change, we have ignored this effect for now, but we note that it may more become significant as larger numbers of policies are processed.

6.1.4) Effect of Re-engineering the Code on Performance

In Section 6.1.3 we discussed the need to re-engineer the code so that the scope of variables did not inhibit the introduction of OpenMP parallelism. In this Section, we consider the effect of re-engineering on all platforms, although again we only performed a summary investigation on the desktop and on Ness. The runtimes were

<table>
<thead>
<tr>
<th>Description</th>
<th>Code Version</th>
<th>Runtime (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>before re-engineering</td>
<td>04m</td>
<td>58.95</td>
</tr>
<tr>
<td>optimisation of ‘unused’ reserve calculations</td>
<td>05m</td>
<td>58.41</td>
</tr>
<tr>
<td>final re-engineered code</td>
<td>06u</td>
<td>57.66</td>
</tr>
</tbody>
</table>

Table 6.1: Time to process 1000 policies during re-engineering

From the results in Table 6.1, we see that re-engineering the code had a minor overall effect on the performance on the laptop and desktop, but the size of the variation was considered insignificant in terms of the overall optimisation, and was not investigated further.

Table 6.1 also shows that propagating the changes to optimise other reserve calculations, as discussed in Section 4.6, has had the effect of improving the performance of the code on Ness. This was unexpected because the routines to calculate the reserves for other types of policy are not called during the processing of the synthetic data used here, and so it is difficult to see how changing those routines affects performance during processing of this data. It may be the case that PGF90 has optimised the code in a different way to IFort, leading to some extremely efficient assembly language being produced and, at runtime, testing whether that efficient code should be used. While use of such efficient code should lead to reduced runtimes, we have not investigated this possibility further.

If we estimate the rate of processing in relation to the core’s frequency, then we obtain

\[
\text{laptop: } \frac{1000 \text{ policies}}{57.7 \text{ sec}} \approx \frac{8.7 \times 10^{-9}}{2.0 \text{ GHz}} \text{ policies} \\
\text{desktop: } \frac{1000 \text{ policies}}{37.9 \text{ sec}} \approx \frac{10.1 \times 10^{-9}}{2.6 \text{ GHz}} \text{ policies} \\
\text{Ness: } \frac{1000 \text{ policies}}{34.9 \text{ sec}} \approx \frac{11.0 \times 10^{-9}}{2.6 \text{ GHz}} \text{ policies}
\]

The similarity of these estimates again indicates that the rate of processing is proportional to the core’s frequency, although there are some minor variations in runtimes between the platforms which are likely to be caused by the different compilers and operating systems. In particular, the fact that the estimate for the laptop is lower than the other platforms may be a result of the number of daemons running on that platform; we deliberately did not spent any time removing unnecessary daemons on the basis that we wanted to imitate the
environment within the business, whereas the desktop and Ness are machines provided by EPCC and these presumably have far cleaner installations of their operating systems.

6.2) Initial Testing of the Parallel Implementation

Having parallelised the code, we need to investigate its performance. There are three types of scaling [17], viz;
Type 1: fixed number of cores, i.e. increase workload without increasing number of cores
Type 2: fixed workload, i.e. increase number of cores without increasing workload
Type 3: fixed workload per core, i.e. increase number of cores to match increase in workload.

Type 1 scaling is considered in Chapter 7 where multi-core systems are used. In the business context there is a fixed number of policies to process, and so Type 3 scaling is not considered in this investigation, although it could be part of a later project. Therefore, in this Section, we only consider Type 2 scaling.

6.2.1) Testing Increasing Number of Cores

So far, the majority of this investigation has used a dual-core laptop as its main platform. In this Section we continue to use that laptop, but solely to test correctness of our OpenMP parallelisation which we implemented in Section 6.1.

Whilst the laptop only has two cores, OpenMP’s scheduling is unaware of how many actual cores are available; OpenMP simply creates a multithreaded program, and the threads are scheduled by the operating system [13]. It is therefore possible to submit programs which are designed to use larger numbers of threads than the number of cores available.

For simplicity, we first looked at the default schedule. What the default schedule actually is depends on the OpenMP implementation [18], and is therefore not necessarily guaranteed to lead to the same performance on all platforms. However, as a test of correctness, it is an adequate first step.

To allow for potential load-imbalance, we also looked at the dynamic(5) schedule. The dynamic schedule allocates chunks of work to threads as each thread finishes its last chunk, thereby ensuring that cores are kept busy. The chunk-size used here was chosen to create sufficient chunks that a smooth load balance might be achieved.

The results for these schedules, using various numbers of threads, were

<table>
<thead>
<tr>
<th>number of threads</th>
<th>Runtime (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>default</td>
<td>dynamic(5)</td>
</tr>
<tr>
<td>1</td>
<td>57.59</td>
<td>57.39</td>
</tr>
<tr>
<td>2</td>
<td>28.92</td>
<td>28.81</td>
</tr>
<tr>
<td>4</td>
<td>28.81</td>
<td>28.77</td>
</tr>
<tr>
<td>8</td>
<td>28.78</td>
<td>28.78</td>
</tr>
</tbody>
</table>

Table 6.2: Time to process 1000 policies using various numbers of threads and different schedules

From these results, it is apparent that, irrespective of how many threads OpenMP has produced, the maximum speedup is 2. This is clearly consistent with the fact that we have tested our implementation on a laptop which only has two cores.

These results also suggest that the different schedules do not produce a significant difference in processing times. However, to allow for the possibility of the default schedule leading to load imbalance, we use OpenMP’s dynamic(5) as our standard schedule for the later parts this investigation which consider increased workload. This has the advantage that we know which schedule is being used on all platforms, so that our timings are consistent; by actively choosing a schedule we have eliminated the possibility of the timings being contaminated by different compilers using different schedules as their default.
6.3) Scaling to Commercial Numbers of Policies

From the results in Section 6.2 it appears that the code scales correctly as the number of cores increases. However, this investigation was conceived in a commercial environment, so we would like the program to be able to process commercial numbers of policies. Therefore, we now consider the changes required to allow the parallel implementation to process this quantity of policies.

6.3.1) Implementation for Commercial Numbers of Policies

The OpenMP code developed in Section 6.1.3 reads a complete file of 1000 policies, and processes all of them before writing all the results. In the commercial environment which inspired this investigation there are roughly 160,000 annuity policies of this type to be processed. To be able to process this number of policies, we introduced an outer loop so that we could repeat the read-process-write cycle, exemplified in Extract 6.1, many times in order that the complete dataset could be processed.

Our initial implementation is limited to cases where the number of policies in the data is an integer multiple of the number of policies processed in each step. To avoid confusion with OpenMP’s ‘chunks’, we refer to the number of policies covered in each read-process-write cycle as a “bite” so that, for example, 1000 policies can be processed in 4 bites, each of 250 policies. This approach is illustrated by the following pseudocode;

```
integer, parameter :: numberOfPolicies = 1000, biteSize = 250

numberOfBites = numberOfPolicies / biteSize

open inFile for input as #1
open outFile for output as #2

do j = 1,numberOfBites
   do k = 1,biteSize
      input #1, policyDetails(k)
   end do
   parallel do
      do k = 1,biteSize
         policyResults(k) = process( policyDetails(k) )
      end do
   end parallel do
   do k = 1,biteSize
      write #2, policyResults(k)
   end do
   close #1
   close #2

Extract 6.2: Pseudocode for processing using multiple bites
```

This approach keeps the advantage that policies appear in the results file in the same order they appear in the datafile, thereby maintaining the reliability of testing.
6.3.2) Correctness of Scaling using Multiple Bites

In order to verify the correctness of our implementation, we processed 1000 policies using various bite sizes. However, to allow for the fact that the OpenMP schedule was `dynamic(5)`, we avoided using very small bites in order to minimise the possibility of load imbalance. The results for a selection of bite sizes were

<table>
<thead>
<tr>
<th>Bite Size</th>
<th>Version</th>
<th>Runtime (sec)</th>
<th>Rate (policies per sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>07q1</td>
<td>28.91</td>
<td>34.6</td>
</tr>
<tr>
<td>500</td>
<td>07q2</td>
<td>28.94</td>
<td>34.6</td>
</tr>
<tr>
<td>250</td>
<td>07q3</td>
<td>29.02</td>
<td>34.5</td>
</tr>
<tr>
<td>100</td>
<td>07q4</td>
<td>30.06</td>
<td>33.3</td>
</tr>
<tr>
<td>50</td>
<td>07q5</td>
<td>30.50</td>
<td>32.8</td>
</tr>
</tbody>
</table>

Table 6.3: Time to process 1000 policies using various bite-sizes

From Table 6.3 it is clear that the processing times are all similar, and that there is some indication that the processing rate, in terms of policies per second, may increase slightly as the size of the bite increases. However, this sample is not large enough to draw such conclusions with any reasonable level of statistical certainty.

The results in Table 6.3 show that our concept of taking multiple bites out of the dataset in order to process the whole dataset is a success; we are able to process the data using varying bite sizes without detrimentally affecting performance. This initial success on the laptop, although limited to a small number of policies, provides a good indication that the code should scale well as the number of policies processed increases.

To investigate this scaling further, we amended the code so that the array which holds the data read in, and written out, is declared within a Fortran `common` block in order to ensure that the array was held on the heap, thereby allowing significantly larger bite sizes to be used. However, in order to be able to use `common` blocks with `iFort` it was necessary to change some of the derived types so that their elements are declared using the `sequence` modifier, this change being greatly simplified by the re-engineering covered in Section 6.1.3.

The runtimes obtained on the laptop, using the `dynamic(5)` schedule with two threads, were

<table>
<thead>
<tr>
<th>Number of Policies</th>
<th>2000</th>
<th>4000</th>
<th>8000</th>
<th>16000</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,000</td>
<td>57.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4,000</td>
<td>113.3</td>
<td>113.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8,000</td>
<td>225.3</td>
<td>225.1</td>
<td>226.3</td>
<td></td>
</tr>
<tr>
<td>16,000</td>
<td>452.4</td>
<td>452.1</td>
<td>452.0</td>
<td>453.5</td>
</tr>
<tr>
<td>32,000</td>
<td>901.2</td>
<td>900.7</td>
<td>900.4</td>
<td>900.4</td>
</tr>
</tbody>
</table>

Table 6.4: Time to process varying numbers of policies and bite sizes

The results in Table 6.4 look favourable because time increases roughly linearly with number of policies. To be consistent with the results in Table 6.3, we also consider the ‘processing rate’ interpretation of these results, viz
Figure 6.1: Processing rate for varying numbers of policies and bite sizes

Figure 6.1 shows that there is only minor variation in the rate of processing, reinforcing the observation of near-linearity of the increase in processing time as the number of policies increases.

These results suggest that our implementation could be used to process far larger numbers of policies without significant deterioration in the processing rate. This will be investigated in Chapter 7, where we consider processing larger numbers of policies, larger bite sizes and more cores.

6.4) Summary

In this Chapter we have discussed the implementation and initial testing of parallelism. A significant part of the effort to implement parallelism was the re-engineering required to place the declaration of variables on the correct threads. This re-engineering was not only beneficial in terms of software engineering, it was essential to get OpenMP working.

The correctness tests in Section 6.2.1 indicate that parallelism works on 2 cores; perhaps the most important test of correctness is that the results from the parallel program are identical to those from the serial program (to the 6 significant figures output), as discussed in Section 1.7.

In Section 6.3, we implemented a procedure which allows our parallel implementation to process data files which contain more policies than the 1000 used in earlier Chapters. The results in Section 6.3.2 suggest that we should be able to process data files with far larger numbers of policies, and this will be investigated in Section 7.1.2.
7) Scalability of the Parallel Code

In Chapter 6, we discussed the addition, and initial testing, of parallelism. In Section 6.2.1, we saw that the parallel code was over 99% efficient on the 2 cores of the laptop, and in Section 6.3.2, we saw encouraging evidence that the code should be able to process the quantities of policies required by the commercial environment. In this Chapter, we discuss the scalability of the parallel code on platforms with greater numbers of cores.

First we consider the parallel scalability of the code by investigating its performance on each platform while it processes the 1000 test polices, but using varying numbers of cores. For this we use the platforms introduced in Section 5.2, i.e. the desktop and Ness, because these both have more cores than the laptop.

We then investigate the performance of the code as it processes increasing numbers of policies. In order to be able to consider the numbers of polices relevant to the business, we had to create a larger data set. However, that was trivially done by re-running the program which created our original data in Section 1.4; we just changed the variable which controlled the number of policies created.

7.1) The Parallel Code

In Section 6.2 we performed initial testing of the parallel implementation, and we discovered that the speedup achieved from multiple threads was almost linear, on the two cores of the laptop. Whilst that was encouraging, the investigation of interest lies in whether the code scales well as more cores are used; that scalability is considered in this Section.

7.1.1) Increasing the Number of Cores

In Section 6.2, we saw that if there were more OpenMP threads than cores, then the operating system schedules those threads so that the maximum speedup corresponds to the number of cores, rather than the number of threads. This suggests that there is no benefit in having more threads than cores, and so in this Section, we ensure that the number of OpenMP threads does not exceed the number of cores the machine has available. Also, to be consistent with Section 6.2, we limit this part of the investigation to using OpenMP’s default and dynamic(5) schedules.

The results for 1000 policies on the desktop were

<table>
<thead>
<tr>
<th>Cores</th>
<th>runtime (sec)</th>
<th>policies per sec</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>default</td>
<td>dynamic(5)</td>
<td>default</td>
</tr>
<tr>
<td>1</td>
<td>37.53</td>
<td>37.75</td>
<td>26.6</td>
</tr>
<tr>
<td>2</td>
<td>18.96</td>
<td>18.92</td>
<td>53.0</td>
</tr>
<tr>
<td>4</td>
<td>9.47</td>
<td>9.53</td>
<td>105.6</td>
</tr>
</tbody>
</table>

Table 7.1: Time to process 1000 policies, using the parallelised program, on the desktop

From Table 7.1 we can see that the speedup is close to linear for both schedules. It is trivial to calculate the efficiencies for these results; we find that the code is at least 99% efficient for all cases. This efficiency suggests that our code will perform reasonably well in Windows environments with large numbers of cores. However, we did not have access to a machine with Windows XP, and a larger number of cores, on which to test whether this suggestion is actually well founded.

Performing the runs on Ness provides slightly more information about the parallel code’s performance, simply because Ness has more cores. The equivalent results for 1000 policies on Ness were

<table>
<thead>
<tr>
<th>Cores</th>
<th>runtime (sec)</th>
<th>policies per sec</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>default</td>
<td>dynamic(5)</td>
<td>default</td>
</tr>
<tr>
<td>1</td>
<td>39.81</td>
<td>40.27</td>
<td>25.1</td>
</tr>
<tr>
<td>2</td>
<td>20.23</td>
<td>20.01</td>
<td>49.4</td>
</tr>
<tr>
<td>4</td>
<td>10.13</td>
<td>10.08</td>
<td>98.8</td>
</tr>
<tr>
<td>8</td>
<td>5.24</td>
<td>5.06</td>
<td>191.0</td>
</tr>
<tr>
<td>16</td>
<td>2.74</td>
<td>2.63</td>
<td>365.6</td>
</tr>
</tbody>
</table>

Table 7.2: Time to process 1000 polices, using the parallelised program, on Ness
The results in Table 7.2 suggest that the speedup is again close to linear for both schedules. However, given the numerical values involved, a graphical representation provides a more intuitive interpretation;

![Speedup on Ness](image)

Figure 7.1: Speedup for 1000 policies on Ness

Figure 7.1 suggests the scaling achieved on Ness is also reasonably close to linear, and that the `dynamic(5)` schedule generates slightly better scaling than the `default` schedule. This is consistent with `dynamic()` schedules having been designed to incorporate a degree of built-in dynamic load balancing.

Actually calculating the efficiency on 16 cores gives

- **default** schedule: \[\text{efficiency} = \frac{\text{speedup}}{\text{number of cores}} = \frac{14.56}{16} = 91.0\%\]

- **dynamic(5)** schedule: \[\text{efficiency} = \frac{\text{speedup}}{\text{number of cores}} = \frac{15.30}{16} = 95.6\%\]

These efficiencies indicate that the code will not perform particularly well when far larger numbers of cores are used, e.g. 256 cores. However, this is not a particular concern because it is unlikely that the code will be run on a shared-memory machine with many more than about 64 cores in the foreseeable future.

Therefore, while we note the potential lack of parallel scalability, we ignore it for the remainder of this project, not least because no shared-memory platform with more than 16 cores was available to this investigation, and so we clearly cannot pursue the extrapolation of this apparent minor inefficiency in the scaling. An investigation into the scaling of the code to more than 16 cores would make a natural basis for a future project; such a project should also consider different schedules on those higher core counts.

By considering the processing rate for polices we can compare the performance of the platforms, but in order to do this we need the OpenMP schedules to be consistent. We noted in Section 6.2.1 that the actual scheduling used in the `default` schedule depends on the OpenMP implementation, and so, to be valid, our comparisons should use the `dynamic(5)` runs.
Taking the ‘processing rate’ values from Table 7.1 and Table 7.2 we obtain:

<table>
<thead>
<tr>
<th>Cores</th>
<th>policies per sec Desktop</th>
<th>policies per sec Desktop</th>
<th>policies per sec per core Ness</th>
<th>policies per sec per core Ness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26.49</td>
<td>24.83</td>
<td>26.49</td>
<td>24.83</td>
</tr>
<tr>
<td>2</td>
<td>52.85</td>
<td>49.97</td>
<td>26.43</td>
<td>24.99</td>
</tr>
<tr>
<td>4</td>
<td>104.92</td>
<td>99.25</td>
<td>26.23</td>
<td>24.81</td>
</tr>
<tr>
<td>8</td>
<td>197.63</td>
<td>197.63</td>
<td>24.70</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>379.94</td>
<td>379.94</td>
<td>23.75</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3: Processing rate, using the \texttt{dynamic(5)} schedule, for 1000 policies

From Table 7.3 it is clear that, for both platforms, the processing rate per core is reasonably stable as the number of cores increases. This suggests that the \texttt{dynamic(5)} schedule is performing some amount of load balancing, which, by the nature of the \texttt{dynamic()} schedules, is what we would expect.

An additional interpretation of the results in Table 7.3 is that our OpenMP implementation of the task decomposition scales almost linearly as the number of cores increases, and this scalability results from using a \texttt{dynamic} schedule with small enough chunks that load imbalance is almost entirely eliminated.

Although it may be slightly less obvious, Table 7.3 also indicates that the rate of processing is proportional to the frequency of the cores in the machines. This can be seen by considering the runtime on 4 cores (since that is the largest number of cores common to both machines). The cores on both machines run at 2.6GHz and the runtimes are 104.92 sec and 99.25 sec, a difference of about 5.7%. The fact that the times are so similar suggests that the rate of processing is proportional to the core’s frequency, and that the variations in runtimes between the platforms is likely to be caused by the different compilers and operating systems, rather than any particular feature of the code.

7.1.2) Increasing the Number of Policies

From the testing in Section 6.3.2 we know that processing a whole dataset using multiple bites is conceptually successful. This investigation was conceived in a commercial environment where there are significant numbers of policies to process, and so we now investigate larger numbers of policies than have so far been used in this investigation.

The parallel scaling characteristics of the code were discussed in Section 7.1.1, and we saw that the parallel implementation does not appear to suffer much deterioration in performance as the number of cores increases. Since the purpose of this Section is to discover how performance varies with an increasing number of polices processed, we have only used the maximum number of cores on each platform to investigate that scaling. Clearly, we could also have used smaller numbers of cores, but the near-linearity of the relationship between number of cores and time, as indicated by Figure 7.1, suggests that investigating smaller numbers of cores would be unlikely to show anything new.

The time to process a certain number of policies is obviously dependent on the number of polices processed, and so it is more illuminating to present the results in terms of the rate of processing policies, i.e. the average number of polices processed per second.

The investigation into scaling with problem size clearly needs to use significantly larger numbers of policies than have been used for the serial investigation. Since the business environment intends to process 160,000 policies on quad-core PCs, we have only tested scalability up to 160,000 policies on the desktop.
The results for various runs using the four cores of the desktop the results were

<table>
<thead>
<tr>
<th>Number of Policies (thousands)</th>
<th>Policies per second for Bite Size of</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2000</td>
</tr>
<tr>
<td>4</td>
<td>107.4</td>
</tr>
<tr>
<td>8</td>
<td>108.1</td>
</tr>
<tr>
<td>16</td>
<td>108.3</td>
</tr>
<tr>
<td>32</td>
<td>108.0</td>
</tr>
<tr>
<td>64</td>
<td>108.0</td>
</tr>
<tr>
<td>96</td>
<td>108.1</td>
</tr>
<tr>
<td>128</td>
<td>108.2</td>
</tr>
<tr>
<td>160</td>
<td>108.3</td>
</tr>
</tbody>
</table>

Table 7.4: Processing rate, as the number of polices processed increased, on the desktop

These results can be presented graphically as

![Processing Rate using 4 Cores on the Desktop](image)

Figure 7.2: Scaling with problem size when using 4 cores on the desktop

From Figure 7.2 it is clear that the actual processing rate varies slightly with bite size; the variation in processing rate is generally between 107.0 per sec and 108.5 per sec, i.e. around 1.4%. The increase in rate from the first point to the second for the larger bites is around 2.5%. These increases may be due to features of the data, and this could be investigated by producing a set of different data and repeating the scaling runs on that data.

Intuitively, we expect an increase in processing rate as bite size increases; for every bite processed there is some penalty in terms of load balance, and taking larger bites results in the need for fewer bites for a given size of dataset, leading to fewer penalties, and hence an increase in processing rate. Our expectation can be strengthened; by doubling the bite size, the number of bites needed, and hence the overall penalty, halves so that each time we double the bite size the increase in performance halves.

Figure 7.2 shows that the processing on the desktop is broadly consistent with this intuitive expectation. With the exception of the case where the bite size is 2000, there is an increase in the rate of processing as bite size increases, and that increase approximately halves with each doubling of the bite size.

The fact that Figure 7.2 suggests that performance is best when using a bite size of 2000 is counter-intuitive, and may be a result of some feature of the desktop leading to the smaller bite size outperforming the other bite sizes.
To test this possibility, we repeated all of these runs using 4 cores of Ness, and the results are presented graphically as

![Processing Rate using 4 Cores on Ness](image)

Figure 7.3: Scaling with problem size when using 4 cores on Ness

From Figure 7.3 we can make several observations:

1) the processing rate when using a bite size of 2000 is, in general, slightly worse than for the other bite sizes, and this is consistent with our expectation, outlined above. The general pattern of larger bite sizes leading to better performance is visible.

2) the overall pattern for the processing rate, being a slight drop from 8,000 policies to 16,000 policies followed by a slight overall increase, is as evident on Ness as it was on the desktop.

3) the overlapping of the lines in the graph suggest that there is a less structured pattern to the timings on Ness. This may be a result of not using a complete ‘back-end’ of Ness for our processing, thereby allowing the possibility that other users were running programs at the same time. We could test this by performing all these runs again while reserving all the cores, but only using 4 of them.

4) there is no large increase from the first point to the rest of the line for the larger bite sizes. This clearly differs from the desktop, and suggests that something unexpected is happening on the desktop.

The observations resulting from Figure 7.3 suggest that the processing on Ness confirms the fact that some features of processing on the desktop should be investigated; since the variation in processing rate is slight, i.e. less than 2.5%, the deeper investigation has been left for a future project.

In shared-memory mode, Ness can be used with up to 16 cores. It therefore seemed reasonable to attempt to process larger numbers of policies to see whether any detrimental effects appear, i.e. are we likely to break the code by processing too many policies while using a large number of cores? By testing up to 640,000 policies we have significantly exceeded the business’s current requirements, and have therefore built in some allowance for future expansion.
The results for various runs using 16 cores of Ness were

<table>
<thead>
<tr>
<th>Number of Policies (thousands)</th>
<th>Policies per second for Bite Size of</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2000</td>
</tr>
<tr>
<td>4</td>
<td>397.8</td>
</tr>
<tr>
<td>8</td>
<td>400.2</td>
</tr>
<tr>
<td>16</td>
<td>398.1</td>
</tr>
<tr>
<td>32</td>
<td>400.3</td>
</tr>
<tr>
<td>64</td>
<td>400.5</td>
</tr>
<tr>
<td>96</td>
<td>400.6</td>
</tr>
<tr>
<td>128</td>
<td>401.0</td>
</tr>
<tr>
<td>160</td>
<td>400.8</td>
</tr>
<tr>
<td>192</td>
<td>405.3</td>
</tr>
<tr>
<td>256</td>
<td>405.0</td>
</tr>
<tr>
<td>320</td>
<td></td>
</tr>
<tr>
<td>384</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td></td>
</tr>
<tr>
<td>640</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.5: Processing rate, as the number of polices processed increased, on Ness

The results from Table 7.5 are more illuminating when presented graphically, viz;

![Processing Rate using 16 Cores on Ness](image)

From the results in Table 7.5 and Figure 7.4 we can make several observations:

1) the processing rate increases marginally with bite size, and the increase approximately halves as the bite size doubles. These two features align perfectly with our intuitive expectation, as discussed after Figure 7.2.

2) for any particular bite-size, the overall variation in performance as the number of policies processed increases is less than 1%. Whilst this suggests that the processing rate for a particular bite size remains reasonably constant, it is clear from the graph that there is a distinct pattern which may be due to either the arrays which hold the input and output data changing size sufficiently that they cross cache boundaries, or particular features of data within those arrays. Discovering which, if either, of these possibilities is the cause of the deviation could be done by performing these runs again with a different dataset; if the peak and trough appear in the same place then it is caused by features of the cache, whereas the peak and trough appearing in a different place suggest features of the data being the cause.
3) the rate of processing 640,000 policies is extremely similar to the rate for 96,000 policies. This suggests that we should be able to process far larger data sets using our ‘biting’ method.

4) the increase in processing rate as the bite size increases from 8,000 to 16,000 is small, and this suggests that there is probably not much benefit in increasing the bite size much further.

These results indicate particularly good scaling with problem size on Ness.

Overall, the results in this Section suggest that scaling with problem size is good when we process the data using multiple bites. It is clear that the parallelised code could be used to process the numbers of policies required in the commercial environment, and there is no reason to suspect that the processing rate will vary much as the number of policies increases further.

7.2) Summary

In this Chapter we have seen that, on shared-memory platforms with reasonable numbers of cores, the parallelised code scales well. The efficiency achieved using the largest number of cores available and the \texttt{dynamic(5)} schedule is better than 99\% on the desktop, and roughly 95\% on Ness. While these values suggest that our code will not scale well to very large numbers of cores without alteration, we should still obtain acceptable performance to the numbers of cores which are likely to be available in SMPs in the near future.

We have also seen that, for a particular platform, the number of policies processed per second, using our ‘biting’ methodology, does not vary significantly as the number processed increases. This suggests that this code could be used to process many more policies than are currently required by the commercial environment which spawned this investigation.

We have seen, particularly in Table 7.3, that the processing rate per core for each platform is roughly proportional to the core’s frequency. This indicates that our code should also perform well on platforms which have larger numbers of slower cores, e.g. a dual quad-core machine where the frequency is 2.0GHz.

There are several avenues in investigation which could be pursued in a future project;

1) from an HPC perspective, it would be interesting to discover whether increase in processing rate is a true function of the bite size, as suggested by Figure 7.4. The usefulness of this increase in performance to the business is only marginal; whether a code processes 405, or 408, policies per second is immaterial in an environment which is used to polices being processed at one per second.

2) this investigation has primarily considered the \texttt{dynamic(5)} schedule on the basis that it includes dynamic load balancing, and the chunk sizes are small enough to permit a reasonably even balance over the cores. A future investigation should consider other schedules in an attempt to find which is most efficient as the amount of work and the number of cores vary.

3) Figure 7.4 shows that the processing rate increases with the bite size, and this apparent relationship naturally leads to a question as to whether there is a limit to the processing rate, and hence to the code’s efficiency.
8) Conclusion

This investigation has considered an optimisation which was conceived in a commercial environment. In this Chapter, we consider the effect of the optimisation on the performance of the code, and the application of this work to the commercial environment.

First, we summarise the progression of speedup that was achieved during the serial optimisation, and then provide an overview of the scaling of the parallel code.

Finally, we consider how the results from this investigation may be applied to the life assurance office which inspired the investigation.

8.1) Progression of SpeedUp

*The Software Optimisation Cookbook* [11] suggests that “90:10” rule applies to optimisation of software, i.e. "the first 90% of the gain takes 10% of the effort, and the last 10% of the gain takes 90% of the effort". This appears to be consistent with our experience in this project; we have achieved a considerable improvement in performance in a reasonably short time.

In this project, we have applied HPC to the novel area of financial modelling, and we have seen that the optimisation principles which HPC would apply to physics, chemistry and engineering are equally applicable in our field. In this Section, we summarise the speedup seen as our optimisation progressed.

8.1.1) Serial Optimisation

We have tuned compiler options and manually optimised dominant routines, and both of these tasks had a beneficial effect. However, we have not touched the minutiae of architecture-specific phenomena such as cache misses, cache thrashing, or misses within the TLB; such tweaks to the code are likely to take far longer than the time available to this investigation, although they might form an interesting basis for a later investigation.

It is clear from Figure 5.3 that the serial optimisation was beneficial on all platforms, and that the laptop benefited most; the majority of that ‘additional’ benefit resulted from the elimination of the lower default level of optimisation used by the compiler.

Whilst the performance gains in Figure 5.3 are impressive, they are contaminated by the differing start positions caused by the compiler’s default settings. Figure 5.4 suggests that, once */fast* was used, the speedup resulting from each subsequent phase had a similar effect on all platforms.

Figure 5.4 confirms the observation, from Section 5.3.2, that the speedup from interpolation is not as great on Ness as it is on the other platforms, but this ‘loss’ is recouped with the optimisation of the reserving calculation. Whether this feature results from Ness being a linux system rather than Windows, or from Ness using *PGF90* rather than *iFort*, has not been investigated further.

Whilst these differences in speedup may be of some interest, we saw in Section 5.3 that the final performance of the code on each platform is governed by the frequency of the cores on the machines performing the processing.

In Section 6.1.4, we saw that re-engineering the code did not have much of an effect on the laptop or desktop, but there was a slight performance improvement on Ness.
All of these points are illuminated well by considering the overall cumulative improvement during the serial optimisation phase of this project, and this can be seen as

Figure 8.1: Cumulative speedup during serial optimisation

Figure 8.1 provides a succinct illustration of the serial optimisation:
1) changing to /fast was beneficial on all platforms, with the laptop benefiting most
2) manually optimising the interpolation routine produced a similar speedup on all platforms
3) manually optimising the reserve calculation led to further speedup, and the scale of the improvement was about the same on all platforms
4) re-engineering did not have a noticeable effect on the laptop or desktop, and only had a minor effect on Ness.

8.1.2) Parallel Performance

The initial testing of our implementation of an OpenMP task decomposition in Section 6.2.1 suggested that our approach to parallelism was correct; Table 6.2 showed that, using both cores of the laptop, our code was 99% efficient.

In order to process the numbers of policies relevant to the business, we implemented an approach in which the data file was processed using bites. Figure 6.1 shows that initial tests of this approach produced an almost constant rate of processing up to 32,000 policies.

In Section 7.1.1, we saw that the parallel scalability of the code, over the numbers of cores tested, was good. In particular, the code was 96% efficient using the dynamic(5) schedule and 16 cores; this level of efficiency is sufficient for business purposes because SMP’s with more than 16 cores are unlikely to be used in the business in the near future.

Figure 7.2, reproduced below as Figure 8.2, shows that there is less than 3% variation in processing rate on the desktop as the number of policies increases.
This consistency in the rate of processing, as shown by Figure 8.2, indicates that our code scales well on the desktop, and this suggests that the business could implement HPC techniques while continuing to use Windows XP as its operating system.

Figure 7.3, reproduced below as Figure 8.3, shows that there is also roughly 3% variation in processing rate on Ness as the number of policies increases.

The consistency in the rate of processing on Ness, as shown by Figure 8.3, indicates that our code scales well to higher numbers of cores. The minor variation in processing rate as bite size changes indicates that our code should scale to large numbers of cores if the optimal bite size and schedule were used.
8.1.3) Overall Improvement

The provisional title for this report was “From Pterodactyl To Ptarmigan” as a reference to the evolution of a dinosaur.

The evolution of our computational dinosaur’s performance is perhaps best illustrated as;

![Progression of Processing Rate](image)

The notes referred to in Figure 8.4 may be summarised as

<table>
<thead>
<tr>
<th>Note</th>
<th>Source Table</th>
<th>Platform</th>
<th>Number of Policies</th>
<th>Bite Size</th>
<th>Number of Cores</th>
<th>OpenMP Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.2</td>
<td>Laptop</td>
<td>1,000</td>
<td>N/A (initial implementation)</td>
<td>2</td>
<td>dynamic(5)</td>
</tr>
<tr>
<td>2</td>
<td>7.4</td>
<td>Desktop</td>
<td>160,000</td>
<td>16,000</td>
<td>4</td>
<td>dynamic(5)</td>
</tr>
<tr>
<td>3</td>
<td>7.5</td>
<td>Ness</td>
<td>640,000</td>
<td>16,000</td>
<td>16</td>
<td>dynamic(5)</td>
</tr>
</tbody>
</table>

Figure 8.4: Progression of processing rate as optimisation progressed

<table>
<thead>
<tr>
<th>Note</th>
<th>Source Table</th>
<th>Platform</th>
<th>Number of Policies</th>
<th>Bite Size</th>
<th>Number of Cores</th>
<th>OpenMP Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.2</td>
<td>Laptop</td>
<td>1,000</td>
<td>N/A (initial implementation)</td>
<td>2</td>
<td>dynamic(5)</td>
</tr>
<tr>
<td>2</td>
<td>7.4</td>
<td>Desktop</td>
<td>160,000</td>
<td>16,000</td>
<td>4</td>
<td>dynamic(5)</td>
</tr>
<tr>
<td>3</td>
<td>7.5</td>
<td>Ness</td>
<td>640,000</td>
<td>16,000</td>
<td>16</td>
<td>dynamic(5)</td>
</tr>
</tbody>
</table>

Table 8.1: Parameters for parallel runs in Figure 8.4

Figure 8.4 reinforces the fact that the final performance achieved by our code would be very good for the business; the time to process 160,000 policies has fallen from about 44 hours to less than half-an-hour using a quad-core machine in SMP mode.

8.2) Application to the Commercial Environment

This investigation has been performed as part of an MSc in High Performance Computing, and has therefore followed a route appropriate to that purpose. Although the investigation was inspired by a commercial interest, we saw in Section 1.2.2 that the valuation system used within the business has several features which make it distinctly different to the programming environments with which HPC is usually associated; not least amongst these is the inability to tune compiler options.

It is therefore not straightforward to translate conclusions drawn in this project directly back to the commercial environment. However, by considering the relevant features of the valuation package, and the performance gains achieved in this investigation, we may subjectively discuss the benefits to the business, and possibly estimate a quantitative performance improvement.
Perhaps the most important discovery is that the default level of optimisation of the valuation system’s compiler appears to be /O2; by changing this default to /fast there will be an improvement to the performance of all programs produced by the valuation system, not just the one emulated in this investigation. From Figure 5.3, it is apparent that the speedup from changing the compiler’s optimisation is roughly 3.7. We would hope that other programs would show a similar speedup if the optimisation used by the valuation package was set to /fast. However, changing the optimisation level is outside the control of the users of the system; such a change would need to be made by the producers of the valuation package.

It is difficult to see how the business can further benefit from this investigation so long as it maintains its use of the valuation package; other performance gains found in this investigation are dependent on the code being a) parallelisable and b) run on systems with many cores. Both of these requirements are outside the reality of the situation in the business, as explained in Chapter 1. However, this report makes a good foundation for a business case to replace the valuation package with bespoke code, at least for some classes of policy.
References


Appendix A: Algebraic Simplification of Lagrange’s Formula

The code in the routine investigated in Chapter 3 is the implementation of the Lagrange’s formula for finding the cubic polynomial which goes through the 4 points \( \{ (x_i, y_i) \}_{i=0}^3 \). The implementation is based on

\[
P_3(x) = f_0y_0 + f_1y_1 + f_2y_2 + f_3y_3 \quad \text{where } f_k = \prod_{m=0 \atop m \neq k}^{3} \frac{x-x_m}{x_k-x_m}
\]

However, in its most recognisable form \([9]\), the formula is

\[
P_3(x) = \left( \frac{x-x_1}{x_0-x_1} \right) \left( \frac{x-x_2}{x_0-x_2} \right) \left( \frac{x-x_3}{x_0-x_3} \right) y_0 + \left( \frac{x-x_0}{x_1-x_0} \right) \left( \frac{x-x_2}{x_1-x_2} \right) \left( \frac{x-x_3}{x_1-x_3} \right) y_1
\]
\[
+ \left( \frac{x-x_0}{x_2-x_0} \right) \left( \frac{x-x_1}{x_2-x_1} \right) \left( \frac{x-x_3}{x_2-x_3} \right) y_2 + \left( \frac{x-x_0}{x_3-x_0} \right) \left( \frac{x-x_1}{x_3-x_1} \right) \left( \frac{x-x_2}{x_3-x_2} \right) y_3 \quad (A1)
\]

This formula is used to interpolate within a life table where, by construction of the table, the arguments are all integers \( \{ \text{e.i. the } l_i \text{'s are quoted only for integral } x \text{'s} \} \). Further, the implementation uses the two integral ages each side of the actual age under consideration.

So, define \( \text{floor}(\phi) = \text{integer part of } \phi \) \{following standard Fortran semantics\}

Further, \( x \) is the actual age for which we require \( l_x \)

So \( x_0 = \text{floor}(x) - 1 \), \( x_1 = \text{floor}(x) \), \( x_2 = \text{floor}(x) + 1 \), \( x_3 = \text{floor}(x) + 2 \)

Let \( x' = \text{floor}(x) \)

Then \( x_0 = x' - 1 \), \( x_1 = x' \), \( x_2 = x' + 1 \), \( x_3 = x' + 2 \) \( \quad (A2) \)

Let \( f \) be the fractional part of the age

Then \( f = x - \text{floor}(x) \) where \( 0 \leq f < 1 \)

i.e. \( f = x - x' \)

Hence \( x = x' + f \) \( \quad (A3) \)

Substituting \((A2)\) and \((A3)\) into the cubic polynomial \((A1)\) gives

\[
P_3(x) = \left( \frac{(x'+f)-(x')}{(x'-1)-(x')} \right) \left( \frac{(x'+f)-(x'+1)}{(x'-1)-(x'+1)} \right) \left( \frac{(x'+f)-(x'+2)}{(x'-1)-(x'+2)} \right) y_0
\]
\[
+ \left( \frac{(x'+f)-(x')}{(x')-(x-1')} \right) \left( \frac{(x'+f)-(x'+1)}{(x')-(x'+1)} \right) \left( \frac{(x'+f)-(x'+2)}{(x')-(x'+2)} \right) y_1
\]
\[
+ \left( \frac{(x'+f)-(x'-1)}{(x'+f)-(x'-1)} \right) \left( \frac{(x'+f)-(x')}{(x'+1)-(x')} \right) \left( \frac{(x'+f)-(x'+2)}{(x'+1)-(x'+2)} \right) y_2
\]
\[
+ \left( \frac{(x'+f)-(x'-1)}{(x'+f)-(x'-1)} \right) \left( \frac{(x'+f)-(x')}{(x'+2)-(x')} \right) \left( \frac{(x'+f)-(x'+1)}{(x'+2)-(x'+1)} \right) y_3
\]
\[
= \left( \frac{f}{-1} \right) \left( \frac{f-1}{-2} \right) \left( \frac{f-2}{-3} \right) y_0 + \left( \frac{f+1}{1} \right) \left( \frac{f-1}{-1} \right) \left( \frac{f-2}{-2} \right) y_1
\]
\[
+ \left( \frac{f+1}{2} \right) \left( \frac{f}{1} \right) \left( \frac{f-2}{-1} \right) y_2 + \left( \frac{f+1}{3} \right) \left( \frac{f}{2} \right) \left( \frac{f-1}{1} \right) y_3
\]
\[
= -\frac{f(f-1)(f-2)}{6} y_0 + \frac{(f+1)(f-1)(f-2)}{2} y_1 - \frac{(f+1)(f)(f-2)}{2} y_2 + \frac{(f+1)(f-1)}{6} y_3
\]
i.e. \[ R_3(x) = (f+1)(f-1)(f-2) \frac{y_0}{-6} + (f+1)(f-1)(f-2) \frac{y_1}{2} + (f+1)f(f-2) \frac{y_2}{-2} + (f+1)f(f-1) \frac{y_3}{6} \] (A4)

The formula in (A4) is recognisable in the final form of the optimised implementation, where \( f + k \) in this equation translates to \( \text{ageAdjust}(1-k) \) in the code, c.f. Extract 3.11.

So, by performing a piecewise optimisation of the code, we have implemented an algebraically equivalent version of the same formula. It should therefore have been possible to perform the algebra first, and then implement the algebraically equivalent code, thereby going straight to the faster version, and hence obviate the need to perform several stepwise optimisations.
Appendix B: Derivation of the Formulae for the Reserving Calculations

In Section 4.2 we introduced the use of the reserve calculation. This Appendix considers the derivation of the formulae used, and although our derivation only provides an overview, it follows [2] closely.

For illustration; consider a life now aged $x$ to whom the assurer undertakes to make a payment at each future time $t$. Suppose that the current annual amount payable is £1 and that there are $m$ payments per year, so that the amount payable at the next payment is $\frac{£1}{m}$, but we allow for escalation at a rate of $e$ per year at the policy anniversary. Suppose, for simplicity, a payment has just been made so that the next payment occurs in $\frac{1}{m}$ of a year and all following payments occur at subsequent intervals of $\frac{1}{m}$ of a year.

Before proceeding, we need some standard notation from the actuarial profession [19]:

a) $\lfloor y \rfloor$ is the integer part of $y$

and b) $p_{x+r}$ is the probability that the life now aged $x$ will be alive at age $x+r$ where $r \geq 0$.

If we allow for an interest at a rate of $i$ per annum then the expected present value of the payment stream just described is

$$a_x = \sum_{j=1}^{\infty} \frac{(1 + e)^{j/m}}{(1 + i)^{j/m}} p_{x+j/m} \quad \text{(B1)}$$

This present value is referred to as the payment reserve factor.

In reality, there are many confounding factors. For example, the distribution of policy anniversary dates throughout the year means that the escalation for an arbitrarily chosen policy is unlikely to happen on the $m^{th}$ payment and, to complicate things further, each payment is unlikely to be exactly $\frac{1}{m}$ from the date at which the calculations apply.

If $f$ is the fraction of the $m^{th}$ ly (monthly, quarterly yearly, etc.) interval between the date at which the calculations are performed and the next payment, and $j$ is the number of intervals until the first escalation, then our expected value becomes

$$a_x = \sum_{j=0}^{\infty} \frac{(1 + e)^{j/m+f/m}}{(1 + i)^{j/m+f/m}} (\frac{1}{m})^j p_{x+j/m} \quad \text{(B2)}$$

There is a whole branch of actuarial science devoted to the estimation of the probabilities of survival [8]. At the simplest level, we choose some period of investigation, which is often three years, and during that time we observe the number of people alive at each age, and also the number of people dying and their age at death. From these observations we obtain $\{q_x\}_{x=0}^{\infty}$, the proportion of people who are alive at integer age $x$ but die before age $x + 1$. We then take some arbitrary number of new-born lives and, by successively applying these $q_x$’s, we obtain $\{l_x\}_{x=0}^{\infty}$, the number of lives expected to have survived to integer age $x$. Since a new-born is age precisely zero, the arbitrary number chosen is $l_0$, which completes the sequence $\{l_x\}_{x=0}^{\infty}$.

We may then estimate the probability of surviving from age $y$ to age $y+s$ to be

$$p_{y+s} = \frac{l_{y+s}}{l_y} \quad \text{where } y \text{ and } s \text{ are not necessarily integers} \quad \text{(B3)}$$

By introducing the $l_x$’s from (B3) into (B2), the reserve factor for our simple payment stream becomes
\[ a_x = \sum_{i=0}^{\infty} \frac{(1+e)^{i+1/m+j/m}}{(1+i)^{j/m+i/m}} l_x^{x+y+j/m} \]  
(B4)

and this simplifies to

\[ a_x = \frac{v^j}{l_x} \sum_{i=0}^{\infty} \left( 1+e \right)^{i+1/m+j/m} v^i l_x^{x+y+j/m} \]  
(B5)

where \( v = \frac{1}{(1+i)^{j/m}} \)

So far, we have only considered the payments to the policyholder. In reality, there are expenses associated with such payments, e.g. bank charges incurred by the transfer to the policyholder’s account, and some apportionment of the admin expenses of the whole company. We need to allow for such expenses by finding their present value. If we make the fairly reasonable assumption that these expenses are incurred at the time the payment is made, and we make a further assumption that inflation will be applied to these expenses at a fixed date, which is generally the end of the financial year, then our derivation of the expected present value per £1 of these expenses becomes

\[ a_x^{RE} = \frac{v^j}{l_x} \sum_{i=0}^{\infty} \left( 1+e \right)^{i+1/m+j/m} v^i l_x^{x+y+j/m} \]  
(B6)

and this is the renewal expense reserve factor. It is important to note that, in (B6), \( k \) is the number of periods until inflation of the expenses first occurs, and this is, in general, different to \( j \), in (B5), which is the number of intervals until the first escalation of the payment to the policyholder.

Things are further complicated by the allowance of expenses relating to the management of funds and investments relating to these policies. Each year, the insurer deducts a fixed percentage of the assets under management as the charge for these management expenses. However, to make the calculations relating to liabilities independent from the calculations for assets, the liability calculation approximates this charge by a fixed percentage of the present value of payments and expenses. At a basic level, this leads to a circular calculation which is caused by the fact that we cannot find the present value of those expenses until we know what those expenses are, but the expenses are a fraction of the present value of the expenses, etc.

However, we can avoid this circulation by repeating the above calculations at a different interest rate. Suppose the investment expenses are taken at rate \( e' \) per annum. Then the rate of income on the assets is reduced by these expenses so that the effective rate of interest \( i' \) is obtained from

\[ (1+i') = (1+i)(1-e') \]  
(B7)

and we then re-calculate the reserve factors above using this reduced interest rate.

It should be noted that the factors calculated at the reduced rate will be higher than the factors calculated using the original rate \( i \) simply because the reduction in interest rate leads to lower future income which must be compensated for by a larger ‘initial deposit’. This increase in reserve factor can be regarded as the amount required to meet the deductions for investment expenses; it is the investment expense reserve factor.

So, using (B7) to obtain \( i' \), we have

\[ v = (1+i)^{1/j/m} \]

and

\[ u = (1+i)^{1/j'/m} \].
Our payment reserve factors, per £1 of benefit payable to the policyholder, are therefore
\[
a_x = \frac{v^f}{l_x} \sum_{t=0}^{\infty} \left[ (1 + e)^{f/m + f/m} v^t l_{x + t + f/m} \right] \tag{B8}
\]
and
\[
b_x = \frac{u^f}{l_x} \sum_{t=0}^{\infty} \left[ (1 + e)^{f/m + f/m} u^t l_{x + t + f/m} \right] \tag{B9}
\]
and, the renewal expense reserve factors, per £1 of renewal expense attributed to the policy, are
\[
a_{x}^{RE} = \frac{v^f}{l_x} \sum_{t=0}^{\infty} \left[ (1 + e)^{f/m + f/m} v^t l_{x + t + f/m} \right] \tag{B10}
\]
and
\[
b_{x}^{RE} = \frac{u^f}{l_x} \sum_{t=0}^{\infty} \left[ (1 + e)^{f/m + f/m} u^t l_{x + t + f/m} \right] \tag{B11}
\]
Appendix C: Estimation of the Number of Power Calculations

Consider the case where the policy has monthly payments:
At $t = 0$, there are 683 payments outstanding, so the reserve calculation performs 683 powers
At $t = 1$, there are 682 payments outstanding, so the reserve calculation performs 682 powers
At $t = 2$, there are 681 payments outstanding, so the reserve calculation performs 681 powers etc, until the end of the projection, where there are no further payments
Hence, the total number of power calculations for all of the payment reserve calculations for this policy is

$$\sum_{n=1}^{683} n = 683 + 682 + 681 + \ldots + 2 + 1 + 0$$

By an elementary result from 'O' level maths,

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

So, total number of power calculations for the payment reserves is

$$\sum_{k=1}^{683} k = \frac{683(683+1)}{2} = 233,586$$

As noted in Section 4.2, we need six reserves to be calculated at each step
Therefore, the total number of power calculations for all reserves is $233,586 \times 6 = 1,401,516$

Now consider a policy where the payments are yearly:
If we assume that payments are distributed uniformly throughout the year, then on average $\frac{1}{12}$th of the policies will have a payment in each month of the year. An equivalent interpretation of this is that every policy has, on average, $\frac{1}{12}$th of a payment every month.
Therefore

- at $t = 0$, there are $\frac{683}{12}$ payments outstanding
- at $t = 1$, there are $\frac{682}{12}$ payments outstanding
- at $t = 2$, there are $\frac{681}{12}$ payments outstanding

etc, until the end of the projection, where there are no further payments
Hence, the total number of power calculations for all of the payment reserve calculations is

$$\frac{683}{12} + \frac{682}{12} + \frac{681}{12} + \ldots + \frac{2}{12} + \frac{1}{12} + 0 = \frac{1}{12} \sum_{n=1}^{683} n = \frac{1}{12} \left( \frac{683(683+1)}{2} \right) = \frac{683 \times 684}{24}$$

Therefore, total number of power calculations for all reserves is $6 \times \left( \frac{683 \times 684}{24} \right) = \frac{683 \times 684}{4} = 116,793$

An analysis of the 1000 policies in the test data shows that 836 have monthly payments, with the remainder having annual payments; there are no policies where payments happen quarterly or semi-annually.

Therefore, the estimated number of power calculations, is

$$836 \times 1,401,516 + 164 \times 116,793 = 1,190,821,428$$
Appendix D: Post Mortem

The original objective of the project was to discover what factors were affecting the performance of a financial modelling code and, where possible, improve its performance. Measured against that objective, our project has been a success.

The original proposal was to perform serial optimisation of the code, and then have the implementation of parallelism and the investigation of scalability as two separate activities. Splitting these activities was natural, and was actually done because it was clearly beneficial to see how much of an effect the separate stages had on performance.

Although our route through this project differed to the proposed workplan, the overall stages of serial optimisation, implementing parallelism and testing scalability are apparent in the report. The main changes to the original plan were:

1) not performing the restructuring which was suggested might be useful as an early stage of the project. While we did need to re-engineer the code, that was a necessity arising from the need to correctly implement OpenMP, rather than a desire to simply rearrange the code. Further, the re-engineering came at a later stage than originally suggested, and followed several stages of serial optimisation.

2) not investigating possibilities for alternative algorithms which the original plan suggested should also be performed at a reasonably early stage. Clearly, our investigation produced a good increase in performance without the need to implement fundamentally different algorithms.

3) we implemented parallelism at a policy level, rather than at the level of the individual payments within each policy as suggested in the workplan. In the processing of each policy, there are stages which either only need to be performed once, or cannot be performed in parallel. Therefore, the approach of parallelising at the level of individual payments would have been far less efficient than the method implemented.

Overall, these deviations from the workplan did not have a detrimental effect on the outcome of this project.

The following table shows the risks identified in our proposal, and the importance attached to those risks:

<table>
<thead>
<tr>
<th>Risk</th>
<th>Likelihood</th>
<th>Impact</th>
<th>Factor</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>losing sight of main aim</td>
<td>3</td>
<td>4</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>loss of development laptop</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>failure to get clarification of IPR</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>failure of EPCC machines</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table D1: Risks identified in the proposal and their prioritisation

Our management of those risks was successful;

1) the project moved in the correct direction; we did not get lost in incidental investigations.
2) the main development platform, the laptop, survived.
3) IPR issues were circumvented before the project progressed too far.
4) EPCC machines did not fail.

Our risk analysis did have one important failing; we did not foresee the loss of two separate weeks work as a result of a physical injury to the author, the eventual outcome of which was the author undergoing a corneal debridement.

Despite the delays, and deviations from the original plan, this project has succeeded; we have identified the factors affecting the performance of the code, and, through serial optimisation and parallelisation, we have significantly improved that performance.