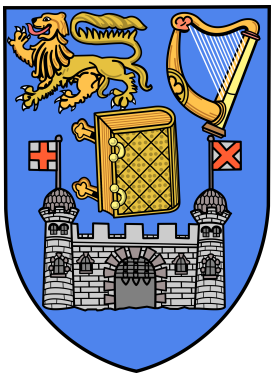
University of Dublin, Trinity College



Animation of Register Allocation via Graph Colouring

Author:

Caoimhe O'Regan

Supervisor:

Dr. David Abrahamson

May 2015

A dissertation submitted in fulfilment of the requirements

for the degree of Master in Computer Science

in the

School of Computer Science and Statistics

Declaration

I, Caoimhe O'Regan, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Signature:

Date:

Summary

Compiler design is currently studied by students in the Computer Science course at Trinity College Dublin. It is divided into two modules, a mandatory one that is taken by all third year computer scientists and a secondary one which is optional in fourth year. In this fourth year module the process of register allocation is studied. Register allocation is the final stage in the compiler pipeline before the code generation phase. It is here where the unlimited amount of registers from the optimization stage are mapped to the actual number of registers available in the machine. The motivation for this work arises from the need to accommodate students in understanding this topic.

The proposed solution to achieve this goal was to create an animation of register allocation which the students could view. The animation was to depict the steps involved to complete register allocation via graph colouring. Graph colouring is one of the methods described in the compiler design module to solve the problem of register allocation. Vivio, a tool for developing E-learning animations for the web, was used to produce the application. Vivio has many capabilities which are vital in creating animations that are easy to use and understand, such as giving the user full control over the flow of the animation and its speed. Each aspect of register allocation via graph colouring is implemented in the application dynamically based on the current program. Each of these stages is then transposed to the screen in the form of an animation.

Once the development of the application was finalised, experiments were carried out to measure the performance of the application with different participants. Thirteen participants were involved in the study and were all enrolled in a computer science related subject. The participants' levels of knowledge on the subject varied and thus gave different insights into the application's performance. The results of the

study revealed an increase in most of the participants understanding of the concept after viewing the animation. This gives some support to the benefits of using the animation to comprehend the subject of register allocation via graph colouring.

The results of this study show there are some benefits in using animations to illustrate the theory of compiler design techniques. Further animations could be developed to demonstrate other procedures of compiler design theory by incrementally developing on this application or creating new ones.

The goal of completing an animation of register allocation via graph colouring has been achieved. This application will be used in the teaching of compiler design in Trinity College next semester to aid students in understanding the approach to register allocation via graph colouring.

Acknowledgements

I would like to thank my supervisor Dr David Abrahamson for his support and guidance throughout the duration of this project.

I would also like to thank my parents for their continued support and encouragement.

Caoimhe O'Regan

University of Dublin, Trinity College

Contents

List of Figures

Chapter 1

Introduction

This dissertation proposes a solution to accommodate students in the learning of a compiler design concept register allocation via graph colouring. It's goal is to complete a totally functional animation depicting register allocation via graph colouring in order to be successful in aiding students in the understanding of this topic.

Compiler design is currently taught over two modules in Trinity College Dublin by Dr. David Abrahamson. The second module covers the topic of register allocation. Register allocation is the final stage of the compiler before the code generation. The purpose of register allocation is to map the variables and intermediate values that reside in an unlimited number of registers (due to the previous optimization stages) into the actual number of registers available in the machine.

In this module Chaitin's [2] approach to register allocation is studied in detail where graph colouring is used to assign variables and intermediate values to the registers available. Chaitin's approach was the first implementation of addressing the problem of register allocation using graph colouring, although many works have tried to improve on this since.

Vivio is a tool created by Dr. Jeremy Jones at Trinity College Dublin for creating E-Learning animations for the World Wide Web [3]. It allows smooth animations to be created that can be completely controlled by the user. It also has the capabilities for playing the animation in reverse. It allows large animations to be created quickly with relative ease and subsequent publishing to the web.

Vivio was used to create an animation of register allocation via graph colouring. The animation displays the algorithm proposed by Chaitin [2] and the underlying implementation also uses the solution proposed by Chaitin. Each of the stages involved are implemented in accordance to Chaitin's description. Some alterations had to be made due to the available structures in Vivio. These stages are then transposed to the screen in the form of an animation. The animation displays each of the stages of the approach in the order of execution, including the transitions between them. Cues are used to draw the users attention to current changes being made and the relationship of the objects on screen.

1.1 Motivation

Many of the students who enroll in the compiler design module have very little knowledge of the theory of graph colouring. The approach itself is easy to comprehend by example, whereas its usefulness in register allocation is a little more difficult to appreciate.

Accordingly, the motivation for this dissertation emerges. By creating an animation that details the steps of graph colouring and its importance in register allocation, it is hoped this will educate and encourage the students of this module.

Furthermore, creating an application that the user can advance through at their own speed could greatly aid in their understanding of the idea. Allowing students access

to this in their own time could further enhance their insight into the concept. It is also well established that learning by example and the use of animation can greatly assist in the understanding of complex ideas.

1.2 Overview of Contents

The structure of this document is as follows;

Chapter 2 *Background*, describes some of the concepts necessary for understanding the application. Each of these concepts is explained with relation to the application and in the order they are utilized in the approach. In addition, some information on E-Learning and the tool used for the animation is discussed. The end of this section includes a description of related work with regard to register allocation via graph colouring.

Chapter 3 *Implementation*, details the approach taken to complete the application. The structure of this section is configured similarly to the previous section where the implementation of each concept is explained in the order of approach. This includes a description of how these ideas were put into effect.

Chapter 4 *Results*, displays the findings of the experiments conducted. Thirteen participants took part in the study. Each of the participants were asked to use the application and fill out a short survey. The results of these are demonstrated in this chapter. Included in this chapter is a discussion of the results and how the experiments could have been improved.

Chapter 5 *Future Work*, expands on the future development of this application. It includes details of improvements that could be made to the application to accommodate other concepts within this topic.

Chapter 6 *Conclusion*, summarizes the project and analyses its goal. Included in this chapter is an overview of the results and some concluding thoughts on the application and its results.

Chapter 2

Background

This section includes some information in relation to compiler design that may be necessary in the understanding this dissertation. This information is related to register allocation and the process of graph colouring.

A description of the tool used for the application, Vivio, including some background information and the reason it was chosen for this project, can also be found in this section.

At the end of this section you will find a discussion of similar works published in this field.

2.1 Compiler Design

The motivation for this project as previously stated, was to assist students of the compiler design II module in understanding the concept of register allocation via graph colouring.

Compiler design is currently taught in Trinity College Dublin by Dr. David Abrahamson and comprises of two modules. The first module is taken by students in third year computer science and is mandatory. This module provides an introduction into compiler design, including an overview of the main principles [4]. Students are introduced to some of the concepts implemented in many compilers and some students are introduced to the theory of a compiler for the first time. A compiler is a program that parses a specific language and manipulates it into machine code to be executed on the machine. Although this may sound elementary, there are many different steps included in order to correctly get from one to the other. These steps are covered in this module, but mainly focus on the opening stages such as parsing.

In fourth year, students of computer science are given the option to enroll in a second module of compiler design. This module is a continuation of the first. The focus of this module is the final stages of the compiler, with it being heavily centred on optimizations and register allocation. It is this module that the application has been developed for. In this module a number of techniques for register allocation are explored. Chaitin's [2], approach to register allocation is studied in detail in accordance to this topic.

2.2 Register Allocation

Register Allocation is a significant phase in compiler optimization. It is used to assign the minimum amount of registers to a large unrealistic number of program variables produced by the compiler optimizations.

During compiler optimizations, intermediate code is generated, the goal of which is to reduce program latency. Thus the intermediate code is generated with the assumption that there are an unlimited number of registers, allowing it to store all necessary variables into fast access memory i.e. registers, rather than having

to load/store to/from memory. This however, is unrealistic for any machine, as all machines have a finite number of registers.

Register allocation is the process by which the intermediate code is translated into a form that uses a realistic number of registers for the machine. This is achieved by allowing different variables to share the same register. The decision of which variables are assigned to the same register is determined by the register allocation phase. This phase is made up of a number of different steps to which the intermediate code is the input and the final code is the output.

Firstly the intermediate code is separated into basic blocks, section 2.2.1. Liveness analysis is performed on the basic blocks in order to find out which variables are live simultaneously, section 2.2.2. An interference graph is created to demonstrate which variables interfere with each other (live at the same time), where nodes represent the variables and adjacent nodes (joined by an edge) interfere with each other. For this project Graph colouring was used for the process of register allocation, so this is the next step and is performed on the interference graph, section 2.2.3. Sometimes it is not possible to colour the graph and so spilling must occur in order to reduce the need for registers, section 2.2.4.

One of the most important aspects of register allocation is that it must not change the behaviour of the program.

2.2.1 Basic Blocks

A basic block is a piece of code that is executed sequentially and has a unique entry point and a unique exit point. To separate a program into basic blocks the leader of each block must be found. The leader is the first statement of each basic block and can be identified by the following characteristics;

1. The first statement of code
2. Any statement that is the target of a GOTO (branch)
3. Any statement that immediately follows a GOTO

Separating a program into basic blocks is a very common practice in compiler design and is used in other aspects of compiler design and not just register allocation.

2.2.2 Liveness Analysis

Live analysis is performed on the program and can be done locally on each instruction or globally on each basic block. Liveness Analysis involves assigning each variable to one or both of two sets for each basic block, namely *LiveIn* and *LiveOut*. To determine whether a variable is a member of the set it must satisfy the equations below which are discussed in the compiler design module.

Live – A variable is live at the end of a basic block if along any, at least one, path from the block it may be used before its value is modified.

$LiveOut(i)$ – A variable is live on exit from a block if it is live on entrance to some successor of the block. If the block has no successor, the last block, then the $LiveOut(i)$ set is empty.

$$LiveOut(i) = \cup LiveIn(j) \quad j \in succ(i) \tag{2.1}$$

$LiveUse(i)$ – is the set of variables that are used before they are defined in block i.

$Def(i)$ – is the set of all variables defined in block i

$LiveIn(i)$ – A variable is live on entrance to a block if it is in $LiveUse(i)$ of the block or live on exit to some predecessor block and was not modified within the block.

$$LiveIn(i) = LiveUse(i) \cup (LiveOut(i) - Def(i)) \tag{2.2}$$

Similar equations can be found in "Compilers: Principles, Techniques, and Tools" [5] for global flow anaylsis. Using the equations above to perform liveness analysis demonstrates which variables are live at a given point in the program. If a variable is live at the same time as another variable they are said to interfere. Variables that interfere are unable to share the same register.

2.2.3 Graph colouring

Graph colouring is a division of graph theory whereby a graph is coloured in such a way that no two adjoining vertices have the same colour. It is commonly known that the problem of graph colouring is NP-Complete [6]. No efficient algorithm has been found and so a more heuristic approach is used.

A graph G consists of pairs (V, E), where V is the set of vertices and E is the set of edges. If a graph is k—*colourable* then we say it can be successfully coloured with k colours. The main goal of graph colouring is to find the minimum number for k such that G is colourable.

In order for G to be k—*colourable* all nodes must be coloured in k or fewer colours and no two adjacent nodes can be the same colour. In regards to register allocation, the different colours represent different registers.

Graph colouring is achieved by following a set of steps set out by an algorithm. For this project, Chaitin's algorithm was used and it is discussed below [2].

The first step is to complete a global data–flow analysis, this was discussed above in section 2.2.2.

From this an interference graph of the variables is created whereby variables in the graph are connected if they are live at the same time, which was briefly discussed above. Each node in the graph is given a degree which represents how many adjacent nodes it has. The next step is to begin the process of colouring the graph.

Chaitin describes the next step where $k = 32$;

> "Assume we wish to find a 32–colouring of a graph G having a node N of degree less than 32. Then G is 32–colourable if and only if graph G' from which N and all its edges have been omitted is 32–colourable."(Chaitin P68)

Each node, with degree less than k, is removed from the graph further decreasing the degree of the rest of the nodes, until all nodes have been removed or no node left in the graph has a degree less than k. The underlying concept is, by removing a node with degree less than k and leaving a graph that is k colourable means that there will be a colour left for this node when it is added back in as it has less than k neighbours.

The nodes are then added back onto the graph, in the reverse order that they were removed, and a colour is assigned to the node that does not conflict with any node it is currently connected to.

Once this process is complete each node in the graph will have been assigned a colour. These colours represent actual available registers in the machine. Variables that have

been assigned the same colour can be stored in the same register at different times. The program can now be rewritten to reflect this.

2.2.4 Spilling

As mentioned above sometimes a *k—colouring* cannot be found for the graph. If this is the case, some variables may need to be spilled in order to reduce the strain on the number of registers.

The need to spill arises when it is impossible to remove a node from the graph as mentioned above since all the nodes have a degree higher than there are registers, i.e. have $degree >= k$. Hence a variable must be selected to reside in memory.

The act of spilling involves adding spill code to the program in such a way that a selected variable is stored in memory and then loaded into a register when it is needed and returned to memory, thus freeing the register as the amount of time the variable is live is reduced. This can help colour the graph, however, adding spill code to a the program can greatly impede the optimizations performed, therefore it should be done sparingly.

Chaitin discusses his approach to spilling in his paper, "Register Allocation via Graph Colouring" [2]. It explains how spilling should be performed and how spill decisions should be made, which are outlined below.

Chaitin states the importance of keeping the graph and program in "step" in order to make appropriate spill decisions. Once a spill decision has been made, the spill code must be entered into the intermediate code and the graph must be rebuilt so that colouring can be re-attempted.

When deciding which register should be spilled, the goal is to insert as little spill code as possible. Spill code increases execution time, hence, choosing a register that

is used often and has to be loaded from, and stored to memory for every use can be very costly. Chaitin's solution to this, is to associate a cost with each node in the graph which refers to the increase in execution time of the program [2].

The following calculation is stated with the assumption that each instruction executes in one machine cycle, and an instruction in a loop executes ten more times than if it were not in a loop [2].

$$H_0(v) = \frac{cost(v)}{degree(v)} \tag{2.3}$$

$$cost(v) = \sum 10^{depth(i)} \tag{2.4}$$

where $i \; \epsilon \; instructions$, v is defined or used in i and $depth(i)$ is the nested level of i (within a loop).

When choosing which node to spill, the goal is to pick the node with the lowest estimated cost found from the equation above where the cost of spilling is divided by its current degree in the graph.

2.3 E-Learning

E-Learning is becoming ever prevalent in today's academic society. For the most part, this is primarily due to universities offering "online courses" to their students as part of their studies.

E-learning however, is not only related to courses provided online but E-learning is defined as "Learning conducted via electronic media, typically on the Internet"[7].

It is therefore not restricted to enrollment in an online course and related to any learning conducted though electronic media which can be done online .

It is recognised that imagery and animation can accommodate in the learning of complex ideas, with the belief that such mediums engage more of the users senses helping them to learn. A number of studies have been conducted in this field. Byrne [8] and Mayton [9] conducted studies on how beneficial animations are to learning where both studies report improvements with animations however, this success cannot be concretely tied to the animation aspect itself. In Byrne it is stated that the animations encouraged learners to predict the algorithm's behaviour rather than accommodating the students learning [8]. In contrast Mayton states that although the impact on learning made by animation of visuals cannot be completely distinguished from that of image cueing, it was shown the use of animation to teach a dynamic process can be beneficial [9].

Furthermore, allowing users to manipulate the speed and flow of the animation can greatly assist their understanding of the concept being displayed. Allowing the user the capability of reversing the animation is very important. In Birch [10] they describe how the professors were unable "to respond to frequent questions arising from students about what had just happened during the animation" without restarting the whole animation, which can be tedious.

The type of animation that will be created for this project is described as a "concept animator" [3] and [10], whereby the animation focuses on a specific topic which in this case is register allocation of compilers.

2.4 Vivio

Vivio is an E-Learning tool that accommodates animations for a web page. Vivio was developed by Dr Jeremy Jones at Trinity College Dublin. The motivation for this tool arose while trying to teach cache coherency protocols to students of a Computer Architecture course at Trinity College Dublin [11].

The goal of Vivio is to achieve the following characteristics as set out in the academic research paper [3];

> "Good educational animations should be easy to install and use, scale with the window in which they are displayed and animate smoothly and with purpose ... The animation speed should be controllable and it should be possible to single-step in both forward and reverse directions or quickly snap forwards and backwards to key points in the animation" [3] P1.

Vivio allows the creation of animations and for these animations to be easily integrated into any web page to be viewed on the Internet. The animations produced respond to user input while still remaining on a direct path. This tool also allows for the reversal of the animation dictated by the user. This will undo any actions that have been done by the user. The interactive nature and the ability to reverse the application allows a user to step through the animation which is an important element of the E–Learning environment.

A Vivio animation is composed of multiple graphical objects displayed on the screen. These objects can be manipulated over time in order to present an idea to the user. Vivio animations are event based and the animation player organises all events by time represented by ticks [3]. Ticks are used for the animation's concept of time. The animation speed can also be defined and/or set by the user in ticks per second. The

Vivio library contains a number of predefined graphical objects such as rectangles, circles etc. and functions that can be performed on these objects making it quick and easy to use.

To accommodate development in Vivio an Integrated Development Environment (IDE) was created. Vivio animations are compiled into compressed vcode files which in turn are compiled into x86 machine code, and finally these are executed to play the animation [3].

The syntax of Vivio is similar to C++ or java and has many of the same capabilities[3]. Vivio is easy to install and run. Vivio currently works on PCs running Windows and is compatible with Internet Explorer, Firefox, Opera, Safari and Google Chrome browsers.

Vivio was chosen for this project because of its capabilities and lightweight nature. The characteristics of Vivio provide the functionality for users to step through the program at their own pace which is essential in an E-Learning software. It will also provide the tools necessary to complete the project without any unnecessary overhead. Vivio will also be familiar to prospective users of this project from their past experience of the course, and so it is hoped they will have a good understanding of how to use the tool without the need for too much instruction.

2.5 Related Work

Chaitin's paper on "Register Allocation and Spilling via Graph Colouring" [2] is well established in the field of compiler design with regards to register allocation. This paper however, is not where the original idea was first introduced. "Register Allocation and Spilling via Graph Colouring" by Chaitin [12] is where graph colouring was first proposed as a solution to register allocation. As both papers remain

very similar they will be discussed together below. Chaitin's first paper "Register Allocation via Colouring" [12], will be referred to as paper one and Chaitin's second paper "Register Allocation and Spilling via Graph Colouring" [2], will be referred to as paper two.

Paper one describes how global register allocation can be performed with the use of graph colouring in an experimental PL/I compiler [12]. Before this paper, this method had never been implemented, although it had been suggested. Their approach is described as "*uniform, systematic and elegant*" in dealing with idiosyncrasies.

In paper one [12] they introduce a data structure namely the interference graph, which is key to the approach. The interference graph has proved to be instrumental in the application of register allocation by graph colouring and is employed by alternative approaches (Briggs [1], Chow [13]) and is discussed again in paper two [2]. In conjunction with this, paper one discusses the essence of interference and what constitutes two nodes in the graph to be adjacent or connected.

An important aspect of the interference graph is how it should be represented [2] [12]. It is necessary to have both random and sequential access to interfering nodes in the graph. The proposed solution to this is described by paper one as follows: To allow for random access a bit matrix is used to represent the interference graph where the indices represent the nodes. For sequential access a list is provided for each node where the list contains references to the nodes it interferes with. This method was reiterated by paper two and has proved to be very successful.

When the interference graph cannot be coloured spilling must take place as previously discussed. Paper one describes a "heuristic and ad hoc" technique for selecting what computation should be placed in memory. Their approach is to reduce the pressure on registers by inserting spill code in parts of the program which are not

executed frequently i.e. outside of loops. Their main criteria for inserting spill code is for computations which are "pass-throughs" of a basic block, meaning they are live at entry into the block but not referenced within the basic block. This is used in conjunction with two rules for inserting the spill code;

> "if a name is spilled anywhere then we insert a store instruction at each of its definition points. And pass-throughs are reloaded according to the following rule: load at entry to each basic block B, every name live at entry to B that is not spilled within B, but that is spilled in some basic block which is an immediate predecessor of B. " [12] P54.

If this method does not succeed then spill code is entered anywhere the register pressure gets too high.

In contrast with this however, is paper two's [2] approach to spill decisions where spill decisions are made based on the cost to the program if a node were to be spilt. The interference graph is supplemented with cost estimates for each node and the node with the lowest cost is to be spilt. Cost estimates for the nodes are established as "the increase in execution time if it is spilled" [2] P100. Cost estimates are calculated on the basis that all instructions execute in one machine cycle and instructions within a loop execute ten more times than if the were not in the loop.

The main goal discussed in paper one [12] is to store as many computations as possible in the available registers rather than memory which is akin to the goal of paper two [2]. It is stated in paper one [12] that when there is no need for spill code to be entered, their proposed method does better than any hand coder would do. Paper two [2] also discusses the success of their approach with respect to the cost in compiler time and the speed of the programs produced, which use registers rather than memory.

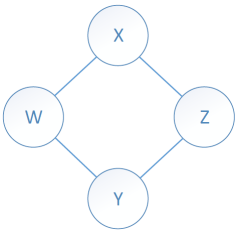


FIGURE 2.1: A simple graph requiring two colours demonstrated by Briggs [1]

Outlined below are other approaches to the problem of Register Allocation. These approaches are formulated on Chaitin's original idea [2]. Both papers discuss Chaitin's idea and how their proposed method improves upon it.

2.5.1 Colouring Heuristics for Register Allocation

Further from Chaitin's heuristic Briggs [1] proposes an improvement. An interesting example used by Briggs is the example shown in figure 2.1. This shows a graph of four nodes each with a degree of two. It is clear to the human eye that this graph can be coloured easily as a 2-colouring i.e. x and y assigned red and w and z assigned blue. Chaitin's algorithm fails to colour this graph with two colours without spilling. Upon first look at the graph there is no node with a degree less than 2 and ergo a node is spilt.

Briggs proposes a solution to this problem by introducing a new way of colouring to enhance Chaitin's algorithm which is based on an algorithm by Matula and Beck [14]. The idea is that instead of removing an arbitrary node from the graph whose degree is less than k, remove the node with the lowest degree. The rest of the approach is completed in the same way as Chaitin's algorithm [2] where nodes are removed until the graph is empty. Once the graph is empty the colouring phase begins where each node is added back into the graph in reverse order, and a colour is assigned to it that has not been used on any of its neighbours.

In contrast to Chaitin's approach the colours chosen are in order and the first one that has not been used is assigned to the node. By using this approach the graph in figure 2.1 will be colourable though, while colouring it may occur that the node's neighbours have already used up all the available colours. When this happens the node is left uncoloured and the colouring of the rest of the nodes is continued. When the entire graph has been rebuilt this method will insert spill code for the uncoloured nodes then rebuild the interference graph and try again [1]. This approach defers the spill decision to the colouring phase which Briggs declares is an improvement on Chaitin's approach and has a higher probability of finding a *k—colourable* for a graph.

The results of this approach proved to be very successful and is stated to be "stronger than Chaitin's method"[1] P288. This technique will colour any graphs that Chaitin's algorithm colours and additionally, some graphs that Chaitin's algorithm does not colour [1]. One issue discussed with this algorithm is that it does not take the cost of spilling into consideration, unlike Chaitin's method. However, a refinement was added to Briggs' technique to incorporate a spill cost. When removing nodes from the graph each of the nodes are ordered based on "cost in those areas where the colouring phase may need to generate spill code" [1] P289 and therefore when removing nodes from the graph it will remove nodes less than k in an arbitrary order. This refinement further enhanced the algorithm so if a node needs to be spilt this method will spill the same node as Chaitin. As a result this method will spill "a subset of the live ranges that Chaitin would spill or the same set" [1] P289.

2.5.2 Register Allocation by Priority-based Colouring

In addition to Briggs' paper another paper by Chow [13] offers improvements on Chaitin's algorithm. As previously mentioned when spilling a variable to memory there is a cost associated whereas this paper introduces a new estimate namely,

saving. Saving refers to the gain in execution speed from allowing a variable reside in a register rather than memory. This is in contrast to the previous techniques which try to store as many variables as possible in registers rather than in memory, whether or not it is beneficial [13]. The proposed solution for this is to assign priorities to variables and assign variables to registers based on this priority.

The algorithm described was used in the production optimizer UOPT [15]. The difference to note here is all program variables are assumed to have been allocated in main memory hence, it is not required to generate spill code but map these references to registers. Chaitin's approach is used with the PL.8 compiler [2] and spill code is generated for variables that can not be allocated a register.

Chow's [13] algorithm is divided into two parts, a local method based on the amount of uses of a variable and a global method which is based on colouring. As previously stated saving costs are established for each variable which is done in the local phase. Two savings are calculated a minimum and a maximum, where the actual saving will range between these two values. Applying a final equation using the minimum cost and the cost to move the variable from memory to the register, and taking into account the number of predecessors and successors, it can be determined if the variable should reside in the register. For the global phase the cost and saving estimates are used along with the depth of a variable access i.e. within a loop. The colouring process is conducted using the cost and saving estimates where registers are assigned a colour based on these estimates i.e. the variables' priorities. As it is assumed variables already reside in memory there is no need to ever insert spill code and the algorithm finishes when all variables have been assigned or there are no registers left.

When colouring if a variable cannot be assigned the same register for the entire procedure, its live range is spilt and the new live ranges are treated as separate variables and are coloured accordingly. If a variable has more neighbours than the

number of colours it is left uncoloured until the end, anticipating that an unused colour can be found for it.

As you can see, this approach is analogous to Chaitin's algorithm. They are based on similar foundations yet implement contrasting approaches. This new approach achieves both "practical and efficient" results which is not effected by the number of registers available [13]. This paper had little influence on this application.

Chapter 3

Implementation

The goal of this project was to complete an animation of Chaitin's approach to register allocation using graph colouring for the purpose of E-Learning. Its aim is to teach students of computer science one of the methods a compiler uses to allocate registers. The application was designed with the objective of educating future compiler design students on the subject matter.

Included in this section, is a detailed description of the application including the underlying structures and the visual animation. This includes the algorithms used; although they are not visible to the user, the underlying application implements all the concepts displayed on screen. Therefore the application itself fully implements register allocation via graph colouring while also displaying the animation to the user.

The implementation of this project is outlined below in two main sections, *Approach* and *Animation*. The first section details the approach used to implement the logic behind the application. The second section details the animation aspect of the project.

3.1 Approach

The application was developed in Vivio, which is detailed in section 2.4. Vivio provided the means to create a detailed animation that can easily be controlled by the user. As Vivio provides similar capabilities to other languages, there was no need to use any other technologies when developing this application. The fundamental logic is computed in Vivio, which will then display the aspects of the computations to the user.

As mentioned, the subject of the animation was Chaitin's approach to register allocation. Although a similar technique was used, due to the capabilities of Vivio and the need to display the process on the screen some alterations were made to the structures and implementation defined by Chaitin. Since the application was implemented with the aim of allowing a user to enter their own program and the application would perform the register allocation on this, a general approach was taken with little to no hard coded values, functions or variables used. This functionality was not included in this phase of the application but is discussed in section 5 as functionality that could be implemented in the future.

These differences are outlined below with reference to the application. Each aspect of the application is discussed in order of execution.

3.1.1 Parsing the program

A program is stored in a single String. In order to be able to parse this program correctly and without a great deal of effort, as this is not the focus of the application, some syntax rules needed to be established. These rules included the following;

1. Instructions must be either one- or two- address instructions

2. Instructions must be separated by a semi-colon
3. ":=" is the assignment operator
4. Only *while* and *if else* statements are supported
5. Statements must start and end with curly brackets

Defining these rules allowed a short concrete algorithm to be constructed that would be capable of parsing any program abiding by these rules.

The procedure passed over each character in the String adding each character to an instruction until it reached a semi-colon or a curly bracket. While this procedure needed to parse the program onto an array of instructions, it also needed to produce a formatted String of the program that could be displayed attractively on the screen. This was achieved by using a count which mirrored the operation of a stack to keep track of brackets for indentation. When an open bracket is encountered the count is incremented, when a closing bracket is found the count is decremented. Before an instruction is concatenated to the formatted program a loop adds indentation equal to the count which represents the nested level. This allows the application to neatly display nested levels of the program to the user that is straightforward to read. The algorithm for parsing the program can be found in appendix A algorithm 1.

A final aspect of the parsing is to separate the instructions into their operands namely, destination, operand one and maybe a second operand. These parts of the instruction are separated from the string based on the definite instruction format. These are used later in the application when performing the live analysis, section 3.1.3 nonetheless, it is completed at this step while the instructions are in a simple array before they are separated into basic blocks in the next step outlined below. These components of the instruction are included in the instruction object as well as a String representing the instruction so that this information will be passed around with the instruction for the entire application, allowing easy access.

3.1.2 Basic Blocks

As mentioned in section 2.2.1 there is a well defined method for separating a program into its basic blocks, but despite this it does not easily transition into code. The difficulty emerges when trying to track the control flow of the graph. In addition to this, for this application each of the basic blocks must be displayed to the user with arrows on screen. Accordingly, an original algorithm was used in the development of this application.

A recursive algorithm was developed in order to account for any number of nested levels. Furthermore the list of characteristics of a leader, section 2.2.1 was redefined to the following;

1. The start of the program as per rule (1)
2. The statement following an *if* statement as per rule (2)
3. The statement following an *else* statement as per rule (2)
4. The statement following a *while* statement as per rule (2)
5. The statement following a closing bracket as per rule (3)

It is apparent the rules haven't changed but have been expanded to the specific cases for this application. More explicitly this algorithm is tail recursive, and as such a structure is passed with each call representing the current state which is employed in the creation of each basic block.

The algorithm works from the top level and is recursively called on each nested level. Each index of the instruction array is tested and if it does not contain any of the statements listed above it is added to the current block's instruction list. If the instruction contains an *if, else* or *while* statement, the current block is closed a

new block is set up in the state object and the function is called on the instructions within the body of this statement. If a closing bracket is encountered, similarly the current block is closed, a new block is set up in the state object and the function returns i.e. returns back from the nested level.

A problem occurs when connecting the *if* block to the block following the *if else* statement. The algorithm returns for the *if* block and then recurses into the *else* part to finally return again. Consequently the current block and previous block have since changed and there is no connection from the *if* part to the following block. To account for this and also any nested *if* statements, the state object being passed around holds a list of *if* blocks that have not been connected to a following block. When there is a block that does not have a next block referenced we pop the last reference off the stack.

While loops cause some difficulty when finding the correct links within the graph. Firstly, the *while* statement itself is removed and instead an *if* statement is placed at the end of the block and the path that should be taken if the *if* statement is true "goes to" the beginning of the loop. It was carefully considered where this should take place and it was decided that it should be done here to ensure the connections of the blocks were correct.

Secondly, when a closing bracket is encountered it needs to be established if this bracket belongs to the end of an *if* or *else* statement or in fact a *while loop*. To accommodate this a separate list is used in the state object. When a *while loop* is detected the instruction is saved to this list and the current block is saved similarly to *if* and *else* statements which are both essentially stacks. Once a closing bracket is found the top of the stack is checked. If it is a *while* statement the saved expression is manipulated into an *if* and placed at the end of the block and is closed, similar to the *if* and *else*. This provides the capability for *while loops* and allows for *if* statements within them.

This algorithm creates an array of basic block objects. Each basic block object records a list of previous block numbers, a list of next block numbers, its own number and the instructions within this block. A pseudo representation of this algorithm can be found in Appendix A algorithm 3. Nevertheless this function only populates the previous block list for each block and not the next block list. The reason for this is when processing each block it is only certain which block came before and it is impossible to predict the full list of succeeding blocks accurately for each block. A separate pass is used in which the blocks are traversed in reverse order and using the previous block information for each block the next information can be added.

3.1.3 Liveness Analysis

For this application iterative global liveness analysis is performed on each of the blocks followed by local liveness analysis on each instruction within a basic block. As previously stated in section 2.2.2 the *LiveOut* and *LiveIn* sets must be computed using equations 2.1 and 2.2 respectively. This analysis is performed from the bottom up.

Global analysis is performed iteratively in order to gain a complete picture of the overall program. This is crucial when handling loops. Consequently, global analysis is performed first and is repeated until the sets converge. The algorithm used for this was similar to that described in [5] and a pseudo representation can be found in Appendix A algorithm 4. Once this is complete local analysis is performed inside each block per instruction, using the *LiveOut* of the block as the last statement's *LiveOut* set. Completing both local and global analysis ensures no interferences are missed.

Using the instruction objects discussed above the analysis can easily perform with access to each variable in an instruction. Once this process is complete variables within the same *LiveOut* set are said to interfere with each other.

3.1.4 The Interference Graph

Chaitin uses two structures to represent the interference graph as outlined in section 2.5, a bit matrix to provide random access and a vector for each node to provide sequential access. Similarly two structures were used for this application however, they are implemented differently.

Firstly interferences are stored for sequential access. Chaitin's approach uses vectors to store the interferences for sequential access, on the other hand Vivio does not support vectors. Vectors are ideal for this use as they can grow as necessary and simply finding the length of a vector for a given node gives the nodes degree.

An alternative solution implemented for this application was to use a linked list for each node stored in an array. For this purpose and other comparison needs in the application each node has a unique number associated with it. This number is used to access the initial index in the array containing the node which will provide $O(1)$ access time to the start of the list. Each node in the list contains two integer values, one which holds the number associated with the node it is representing, and one which holds the index to the next node in the list, or zero if it is the end.

The first node in each list is located in the index corresponding to the associated number of the node. As there is no data which needs to be stored in this node it holds the degree of the node i.e. the length of the linked list. This structure may not seem as elegant as Chaitin's solution but still provides all the same functionality with similar time and space complexity. It also caters for a varying number of nodes

in contrast to Chaitin's structure as it is not necessary to define a vector for each node at the beginning, it will be dynamically assigned space in the array.

Using the information collected in the sequential access the structure for random access is populated. Instead of using a bit matrix for this application a 2D array was used with a type of graphical object which in this case was a line. If there was an interference between two indices the position in the array would contain a line that connected the two nodes. These lines are then used in displaying the graph on the screen. Random access is still provided, though instead of testing if the bit is 0 or 1 there is a test to see if the index contains an object or not. Furthermore, this design only uses half of the structure split down the diagonal while the rest remains empty [12].

Chaitin's modus operandi uses a two pass algorithm over the IL program to first populate the bit matrix then the vectors. This approach populates the interference lists first and then uses this to populate the matrix. Providing random access to interferences is to aid in the completion of coalescing, which is currently not included in this application, however, it could be integrated as part of future work with little effort as both types of accesses need for it [12] are provided. Currently the main function of the matrix is to store the line for the animation as it provides simple access to the lines and the nodes they connect.

3.1.5 Graph Colouring

As outlined above in section 2.2.3, graph colouring is performed on the interference graph which in turn assigns registers to the variables. For this application a variation on Chaitin's [2] approach was used with influence from Briggs [1].

In Briggs [1] it is mentioned that selecting the node with the lowest degree may be a better approach rather than selecting an arbitrary node less than k [2]. Equivalently

this application finds the node with the lowest degree from the interference list, and if this is less than k then this node is removed from the graph. If the node with the lowest degree is not less than k then spill code must be entered, section 3.1.6. As you can see this is a combination of the two algorithms. When a node is removed from the graph it is pushed onto a stack and all interferences with that node are removed from the graph. This is repeated until the graph is empty or spilling must occur.

Once the graph is empty colouring can begin. Nodes are popped off the stack which will be in reverse order, then added back into the graph which is identical to Chaitin's technique. Where the two techniques diverge is in the selection of colours. Chaitin states that when a node is added back into the graph a colour is "picked for it" [2] P99. This is slightly ambiguous for implementing as the only constraint is that it should not be the same colour as any of its neighbours. This notion executes a similar concept as mentioned by Briggs [1]. An array is used where each index represents a different colour. Furthermore, at each index the actual colour value for each colour is stored for displaying on screen. When a node is being added back into the graph the lowest available index is found and this colour is assigned to the node. This is done by starting from the lowest index and finding the first available index that has not been used by any of its neighbours.

When the graph has been coloured the final assignment can take place. Each colour represents a register, meaning if two variables are the same colour then they do not interfere and can reside in the same register. The original formatted program is traversed and each variable is converted to a reference to the register in which it resides. These registers are indicated with "R" concatenated with the index the colour was stored at in the array. This is the final stage of the application.

3.1.6 Spilling

As previously explained, if it is impossible to remove a node from the graph due to all the nodes having a degree greater than or equalled to k, then there is a need to spill. Spilling is discussed in section 2.2.4 where the equations for calculating the spill costs are ascertained at equation 2.3.

Chaitin's procedure for finding the node that should be spilt uses a table to supplement the graph with the cost for each node, equation 2.3. Equivalently, this application supplements the graph with an array to hold the costs for each node. As previously discussed, each node has a unique number associated with it. This number is used to find the corresponding cost in the array. The cost for a given node is the sum of all references where a reference is defined as 10 to the power of the nested level of the reference.

When a node cannot be removed from the graph a heuristic is calculated for all remaining nodes in the graph. This is done by dividing their cost by their current degree. Similar to Chaitin's method the node with the lowest cost estimate is chosen for spilling. The algorithm used for selecting which node to spill can be seen in appendix A algorithm 5.

Once a node is selected for spilling spill code must be added for the node and the process needs to be started again. This is done in order to keep the program and graph in step [2]. Spill code is added at every definition and use of the variable. As the variable is now stored in memory, for each definition the memory location must be updated and for each use the variable must be loaded from memory. This in turn will reduce the live ranges of the variable.

The whole spilling process may need to be repeated several times until a colouring can be found for the graph. Chaitin discusses some "local intellegence" [2] P100, used when making spill decisions however, these were not included in this application.

FIGURE 3.1: First screen displayed to the user

3.2 Animation

Vivio was used to create the animation. Vivio is a tool for creating animations for the web [3] and was discussed in section 2.4. As previously stated, the process of register allocation needed to be fully implemented in order to accurately display the information to the user. Once each step is performed the corresponding animation is displayed to the user keeping the animation as smooth as possible.

3.2.1 Design

The design of the overall application was kept similar to previous animations created and used within the college. This includes the styling and the buttons displayed to the user. The first screen displayed to the user can be seen in figure 3.1. The three leftmost buttons at the top right of the screen are used for controlling the animation. Although the animation can be controlled by using the buttons on a mouse these are provided in case the user does not have access to a mouse, and also to keep the format consistent with other animations.

The start button can be clicked to start the animation. When the animation is playing this button displays the word "stop" and can be used to stop the animation at any time. The reset button can be used to reset the animation to the beginning at any time throughout the animation. The ticks per second button displays the speed at which the animation is currently playing. Clicking this button with either the left or right mouse button will increase or decrease the current speed in a step of ten.

The rightmost button "Prog=1" is the current program being displayed. There are three sample programs included that the user can view by clicking on this button. The first is a simple program that calculates the Nth Fibonacci number, the second calculates the sum of all the uneven numbers less than 10 and the third is a program containing an *if* statement, which produces a complex graph. These programs were chosen because they would be familiar to the user and thus make it easier for them to understand the process. Three programs are included to show how the process works on a variety of examples without overwhelming the user.

The button marked with "K=4" can be used by the student to change the number of k or registers for the program. This can be set to 3, 4 or 5 by clicking this button. These numbers were chosen in order to fully expose the user to the influence this number has. When $K = 4$ which is the default, the graph can be coloured and the registers assigned to the nodes. When $K = 3$ there are not enough registers available to colour the graph and spilling must occur. This will show the processes of spilling to the user. When $K = 5$ the graph can again be coloured. What is interesting about this example is that the application uses a method mentioned by Briggs for choosing which colour to assign to a node, discussed above. The first available colour is chosen for a specific node and as such, this example only uses four out of the five colours.

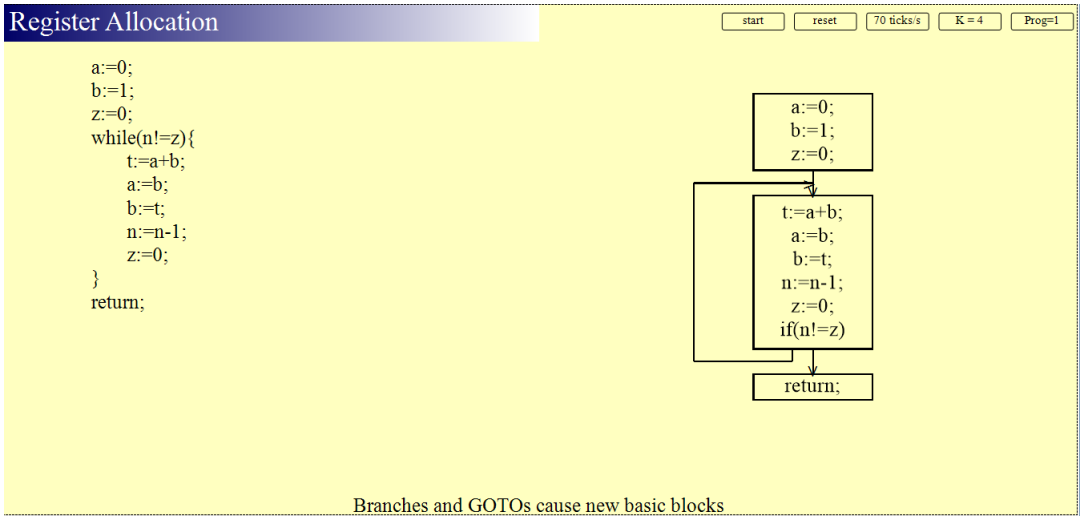


FIGURE 3.2: Program on the left and the Basic Blocks created on the right

3.2.2 Animation Flow

The animation flows in a consistent manner where each new aspect of the process is introduced incrementally so as to not overwhelm the user. Each new step of the process appears on the right hand side and slides to the left after the previous step shown on the left has finished and has disappeared. This will allow the user to anticipate where the next information will appear and help them in understanding the application. An example of this can be seen in figure 3.2 where the program is on the left and the basic blocks created from that program have appeared on the right.

Another mechanism to guide the user through the animation was with the use of flashing. Flashing the background of an object draws the user's attention to what is changing and what it relates to. This will aid the user in understanding the animation and in turn the concept being explained. Flashing was chosen in this context to be consistent with all the other Vivio animations. Although this is the case, this application and all other Vivio animations may need to reconsider this cue. If a student was to have a condition such as photosensitive epilepsy this flashing could be an issue, which was highlighted at the demonstration of the application by the

second reader. For this reason another alerting mechanism should be used such as highlighting the object by simulating a pulsing effect i.e. growing and shrinking the object slightly.

The speed of the animation is initialised to 70 ticks per second. This can be modified by the user to speed up or slow down the animation. This speed was chosen as it seems like an optimal speed to view the animation and it also allows the user to speed up the animation a great deal if necessary.

If spilling occurs, after the spill code is inserted the process must be restarted from the beginning. As both the program and graph need to be kept in step each step of the method must be recalculated in order to generate the correct graph. As these steps have been viewed by the user already this portion of the animation is set to 200 ticks per second to quickly display the process again without forcing the user to watch it all again slowly. Notwithstanding, the user can still slow this aspect of the animation down extensively at any point if they so wish. When the animation reaches the colouring phase again it resumes a speed of 70 ticks per second.

It can be seen from the implementation chapter above that there are a distinct number of steps involved in the process of register allocation via graph colouring. Vivio allows the user to jump through the key parts of the animation. These are known as checkpoints. Checkpoints have been added to the start of each of the main steps in the animation so users can quickly advance through the animation to any step they wish.

The full flow of animation can be seen in figure 3.3. Each screen in this flow indicates the important transitions in the animation. Every screen in the animation is not included as this would be unreasonable.

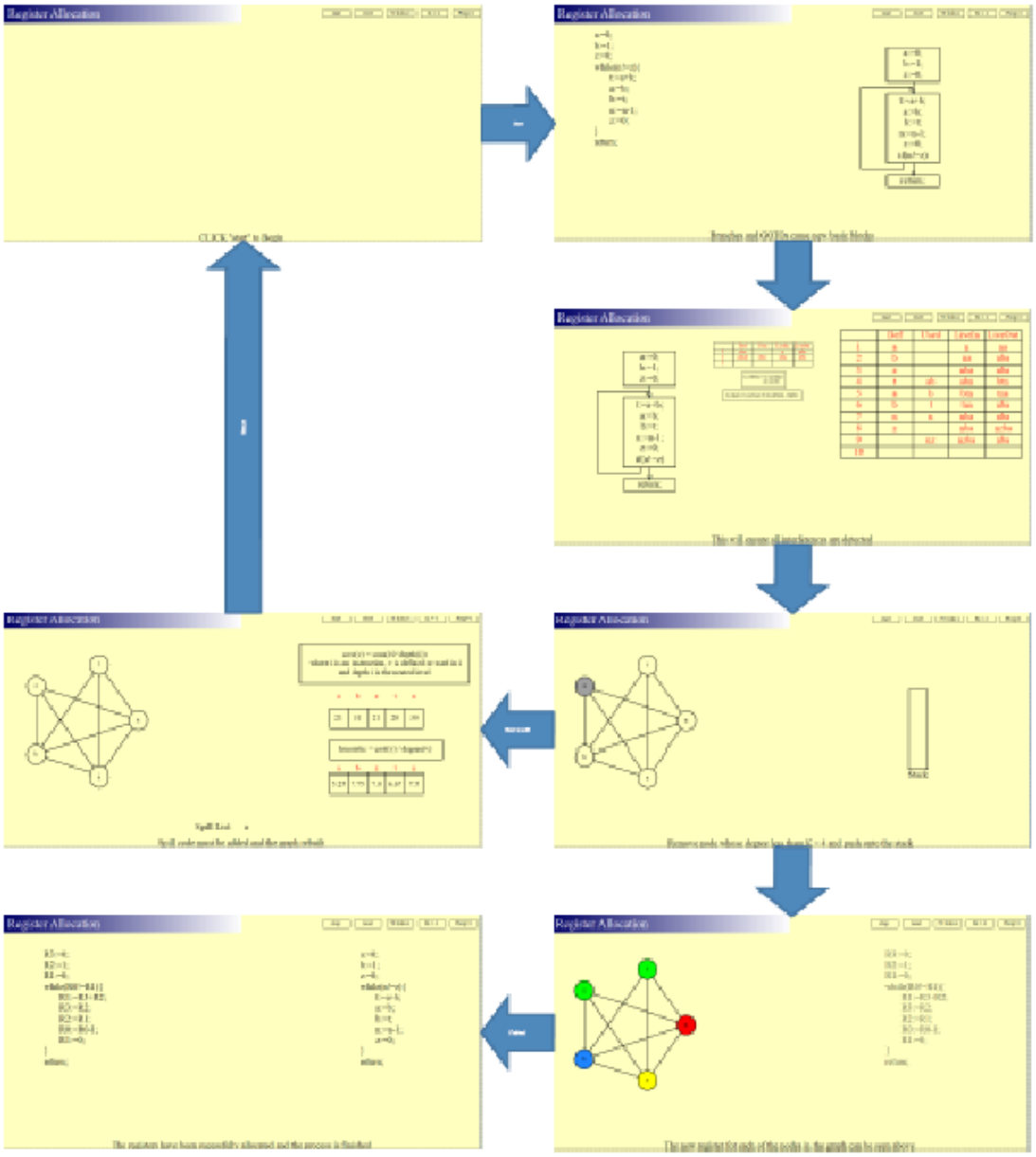


FIGURE 3.3: Complete animation flow

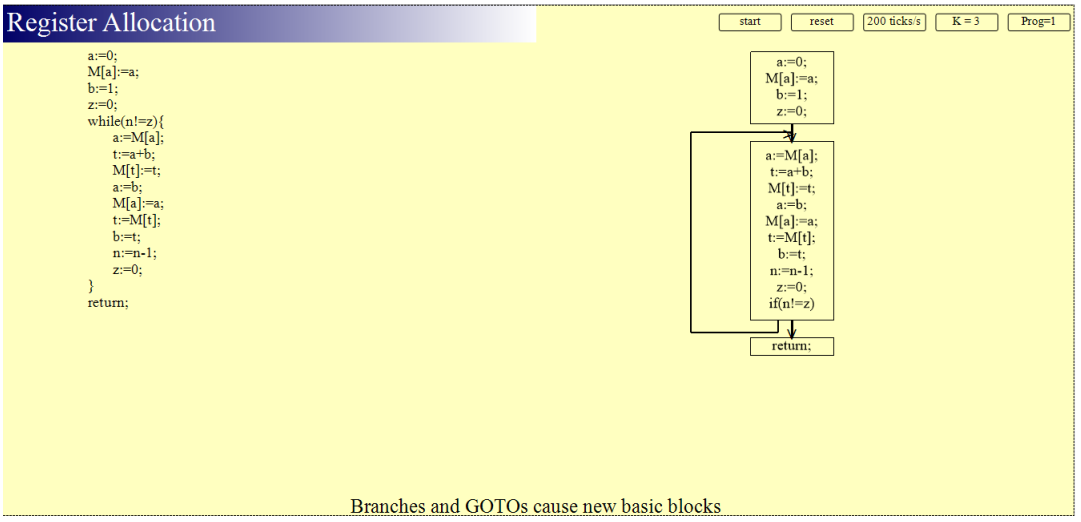


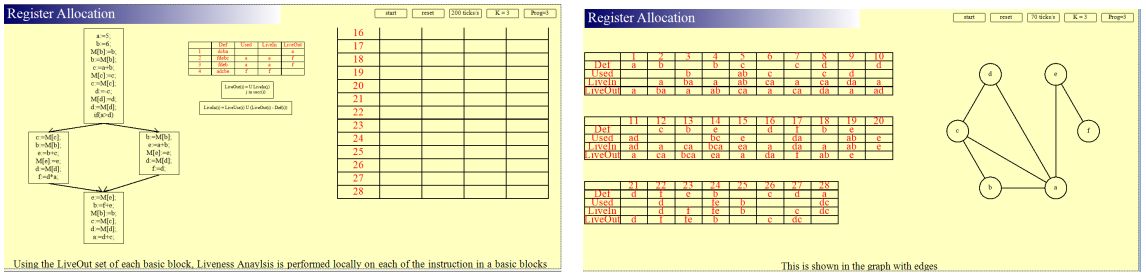
FIGURE 3.4: Scaled program will spill code

3.2.3 Display

When designing the animation particular consideration had to taken with regard to the display. The animation was developed in accordance with an average sized laptop screen utilizing all available screen space. The goal was to distinctively display all aspects of the process, keeping it uncluttered while still being totally visible.

One difficulty encountered was the growing nature of the program after the need to spill. As spill code is added to the program it becomes larger putting pressure on screen space. Some of the examples used can spill numerous times before being colourable and consequently the program, basic blocks and liveness grid can become large enough to flow off screen. To accommodate this each of the animated objects are scaled in accordance to how many variables have been spilt at a given time. This is done by subtracting a fraction multiplied by the amount of variables spilt from 1 and scaling by this factor. This can be seen in figure 3.4 which is the same program as figure 3.2 but with two variables spilt and spill code added for these.

Not to mention this scaling can only be done so many times before the user cannot make out the program. One issue was with the grid displaying the liveness analysis



(A) Scrolling can be seen on the left (B) Table must then be split

FIGURE 3.5: Effects when the program becomes too large

which became too small to see for the example program three, as too many variables had been spilt. To account for this, after a certain number of variables have been spilt no more scaling will take place but the liveness grid will instead scroll while it is being populated. This is shown in figure 3.5 A. Once the table has been filled it is still impossible to display it all on screen for building the interference graph. In order to display the table on screen it needed to be transposed and split into three sections. This is shown in figure 3.5 B.

Chapter 4

Evaluation

The objective of this application was to assist students studying computer science with the topic of register allocation via graph colouring in compiler design. Therefore, it was important that the application not only work but was easy to use and understand. The application must provide the user with detailed information on the subject in order for them to grasp the concept while also encouraging them to learn from it.

This section details the experiments conducted with regards to this application. A description of the study can be found below. The results are shown and a discussion of these results follows.

A study was conducted to determine how the application would perform while teaching students on the topic. A total of thirteen students participated in the study. Each of these students is currently enrolled in a computer science related course in Trinity College Dublin.

The study comprised of two elements, firstly each participant was asked to use the application for as long as they wished. In this time they were free to view the animation at their own pace and also examine other examples by either changing

the value of k as discussed in section 3.2, or viewing other example programs. When they were satisfied with what they had seen they were asked to complete the second element. The second element was a survey which consisted of eight questions. A copy of the survey can be found in appendix B.3.

The survey was given to the participant at the beginning of the study along with an information sheet about the project and instructions on how to use Vivio which are also included in appendix B. Participants were asked to fill out the questions on the survey either before or after using the application with no strict indication of what should be filled out when.

During the course of this study participants were asked to be as honest as possible when completing the survey. It is a well known fact however, that humans behave differently when being watched. Moreover, as mentioned in "Experimental Design from User Studies to Psychophysics" [16] participants in a study may be inclined to provide the answers they think are expected. Students who know the matter being studied and the results that are hoped for can end up consciously or unconsciously providing the answers in a pattern that would match the expectations of the study [16]. This book [16] discusses some solutions to "response bias" however, due to the style, size and time available for this study the solutions were unrealistic for these experiments.

It was hoped these experiments would give some insight into the success of the application. The results of these experiments are outlined below although, as mentioned due to the proximity and subject of the study they should be evaluated while taking *response bias* into consideration.

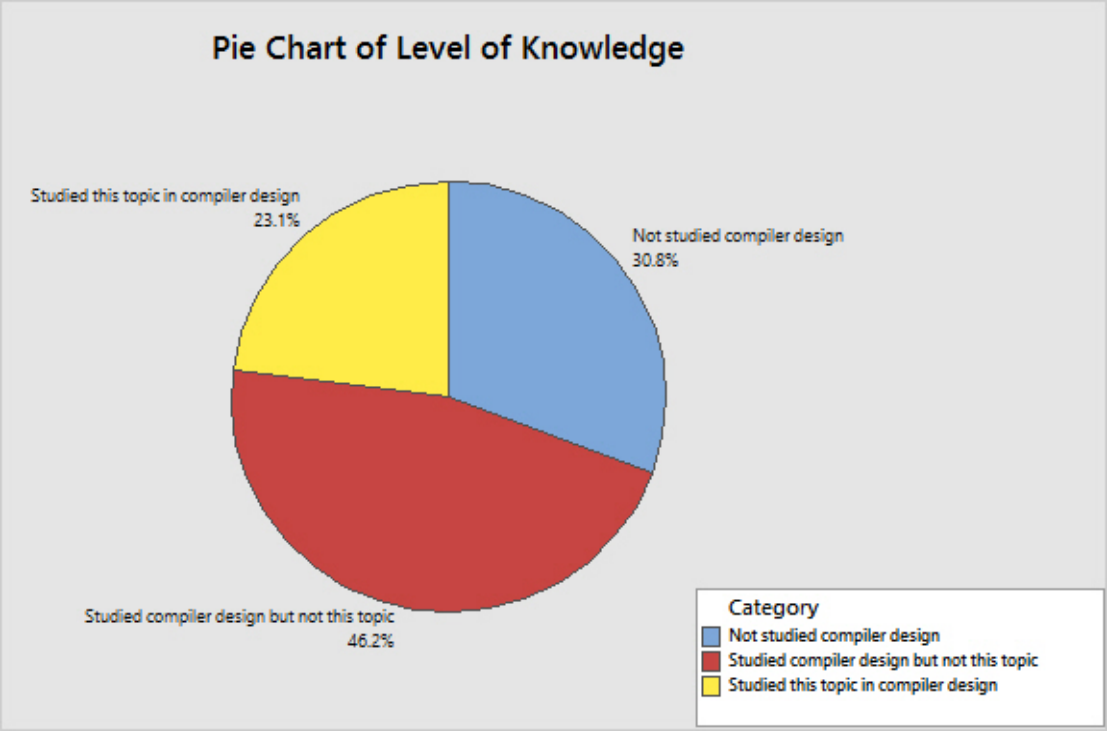


FIGURE 4.1: Different levels of participants

4.1 Results

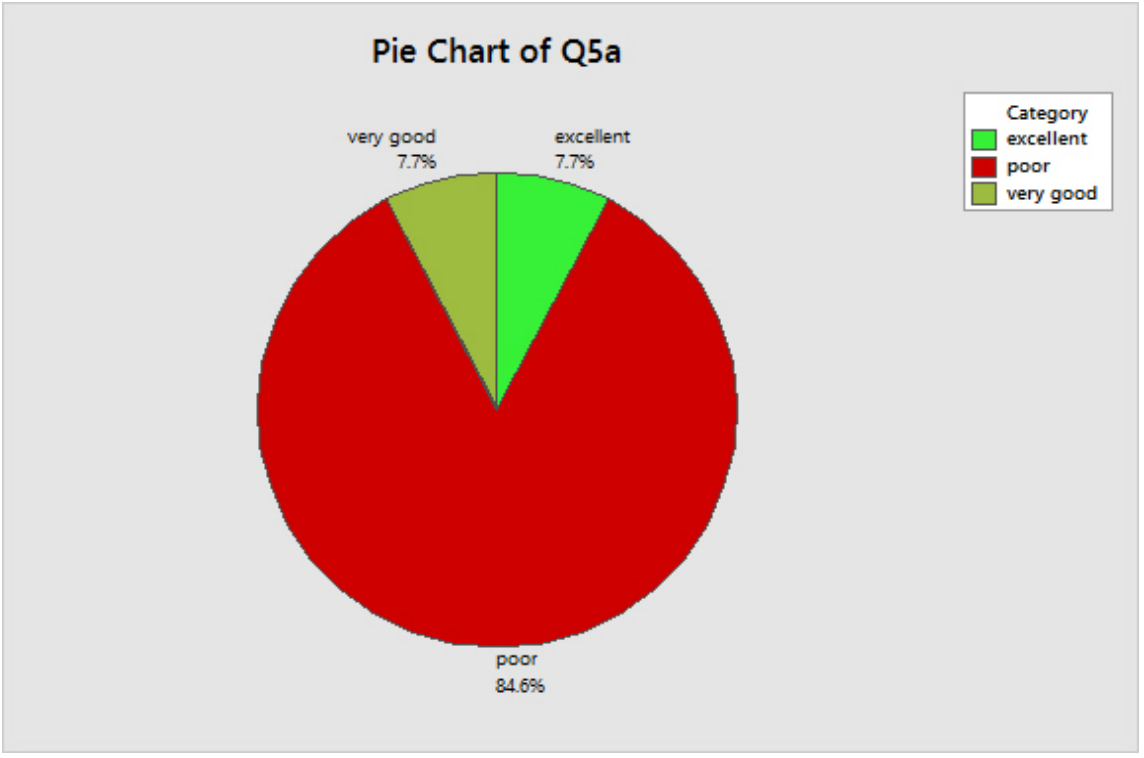
The participants of this study included the following;

1. Students who had not previously studied compiler design
2. Students who had studied compiler design but not this topic
3. Students who had studied this topic in compiler design before

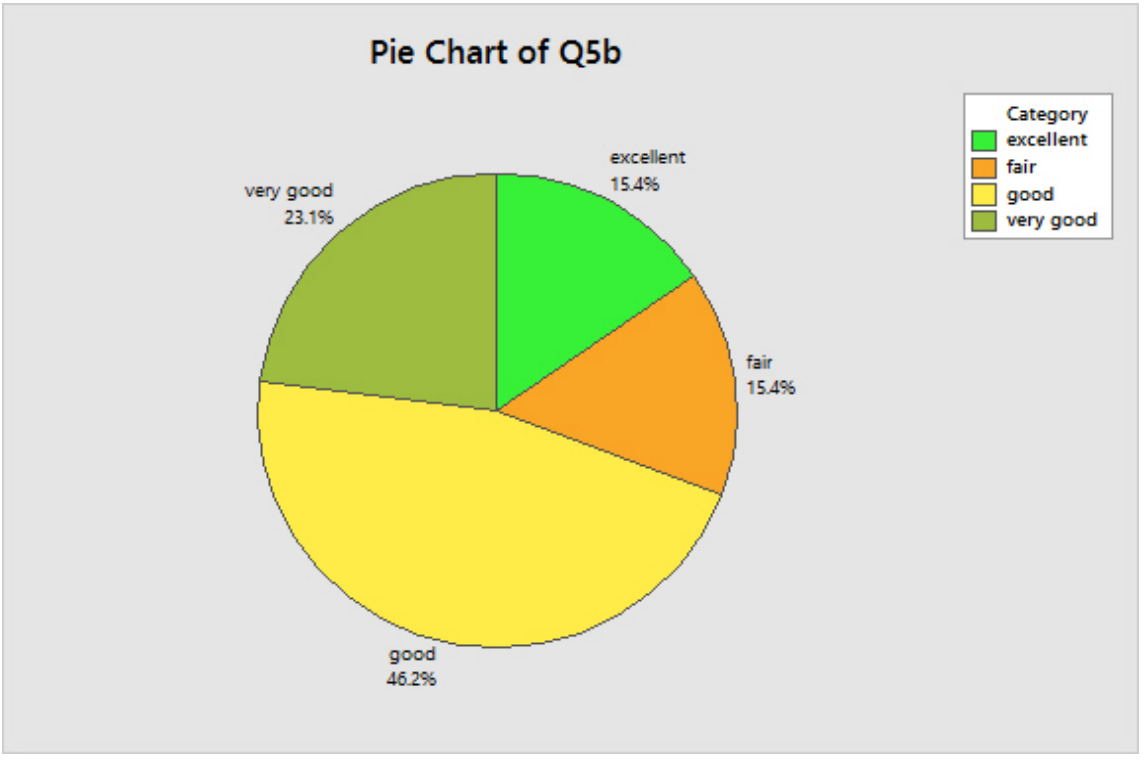
The breakdown of the different groups can be seen in figure 4.1.

The three categories above were included in the study in order to observe the results of participants with different degrees of knowledge on the subject and to see how the application would perform with each.

Ten participants did not know what graph colouring was before the study, this includes participants who had studied compiler design before.



(A) Before using this application



(B) After using this application

FIGURE 4.2: How would you rate your knowledge

Question five of the survey asked the participants to rate their knowledge of register allocation via graph colouring before and after using the application. The results from this can be seen in figure 4.2. The scale used for this question was; poor, fair, good, very good and excellent.

It is clear that before using the application many participants rated themselves as poor, in spite of that after using the application none of the participants classed themselves as poor in fact, a large portion stated their knowledge was now good or higher.

Participants who answered "no" with regard to knowing what graph colouring is all about rated themselves poor in the subject of register allocation via graph colouring. This is to be expected. Be that as is it may, participants who knew what graph colouring was, fell into three categories namely poor, very good and excellent with one participant falling in each group. Only two of the participants who understood what graph colouring was all about knew how it was used in register allocation.

Question two on the survey asked participants if they had an interest in compiler design and these results can be seen in figure 4.3. Nine of the participants did have an interest in compiler design even though many had never studied it before. These results are broken down further and can be seen in figure 4.4.

Figure 4.4 displays the results for question two based on the answer the participant gave for question one, namely *Are you, or have you ever studied compiler design?*. Half the students who had not studied compiler design had an interest in the subject. Additionally, two of the students who had studied compiler design had no interest in the subject.

Everyone involved in the study found the application easy to use and they enjoyed learning about register allocation via graph colouring in this way. They would recommend the application to others.

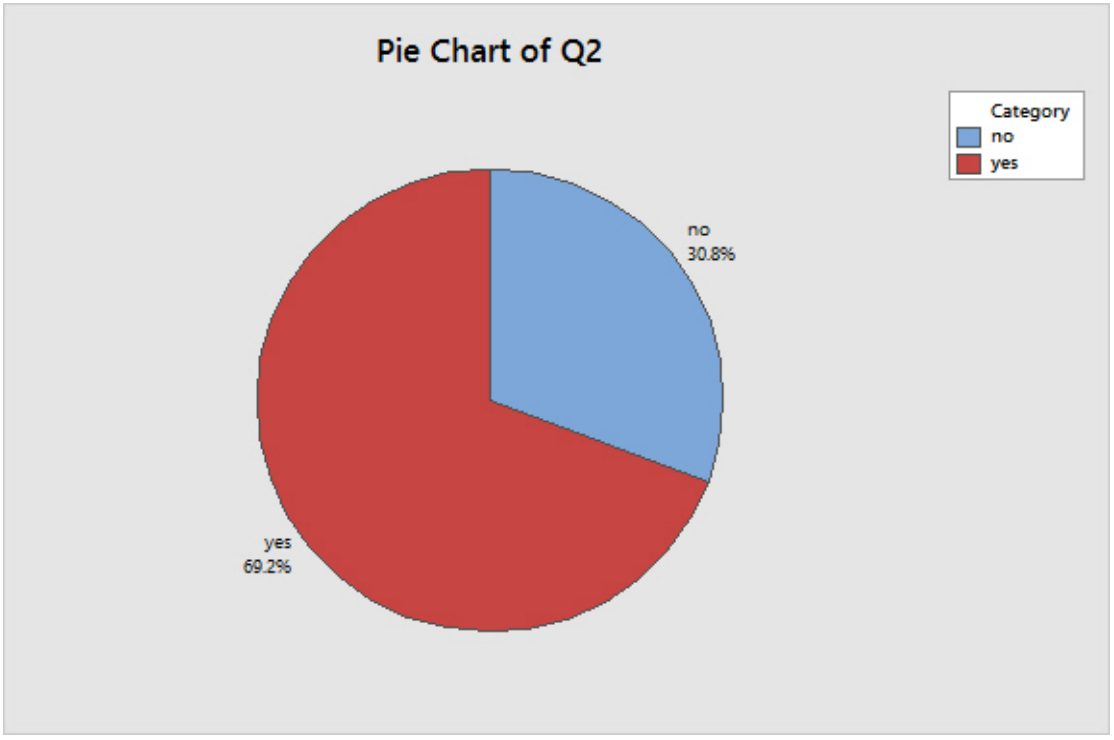


FIGURE 4.3: Q2. Do you have an interest in compiler design

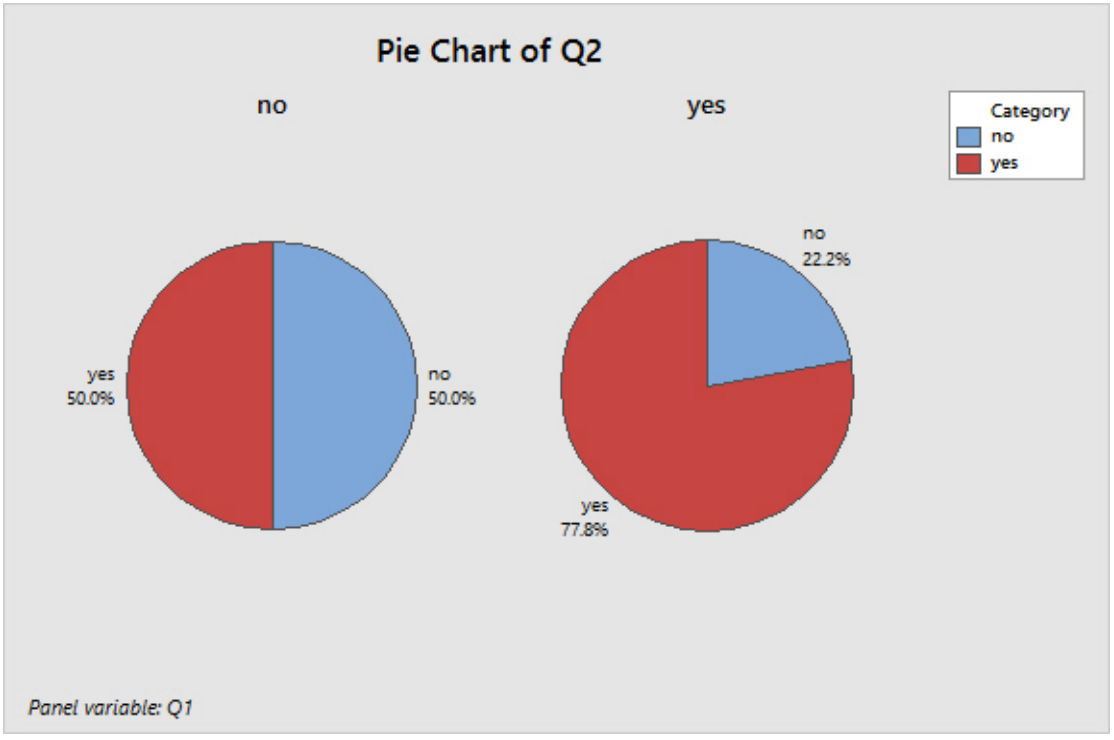


FIGURE 4.4: Question 2 subdivided into the answers for question 1

4.2 Discussion

The results outlined above display some interesting characteristics. A discussion of these results follows.

When examining these results the most important point to consider is the different levels of knowledge the students involved possessed shown in figure 4.1.

It was important to include students at different levels in the study in order to gain a complete understanding of the performance of the application. The application may be easy to understand for students who have studied the topic before but not necessarily for students who have never encountered register allocation or graph colouring. Consequently, students who had studied these topics were included in the comparison and more importantly in the study to find if they enjoyed the application, and of course to see if they could follow the animation easily with their background knowledge.

It was discovered that students who had no previous knowledge in the subject needed some extra information at the beginning of the experiment. This was done verbally by the researcher. This information was given in accordance with the knowledge they would have received in the compiler design module. As this application was developed for use in the class of compiler design and not necessarily as a separate entity divulging this information to the students was in keeping with the parameters of the experiment. The information included details of what register allocation was about and a little information on graph colouring.

Taking the different levels of the students into account the next chart can be analysed. Figure 4.2 demonstrates how the students judged themselves on their knowledge of the subject both before and after using the application. This is one of the most interesting graphs which produced the most relevant results. There is a marked

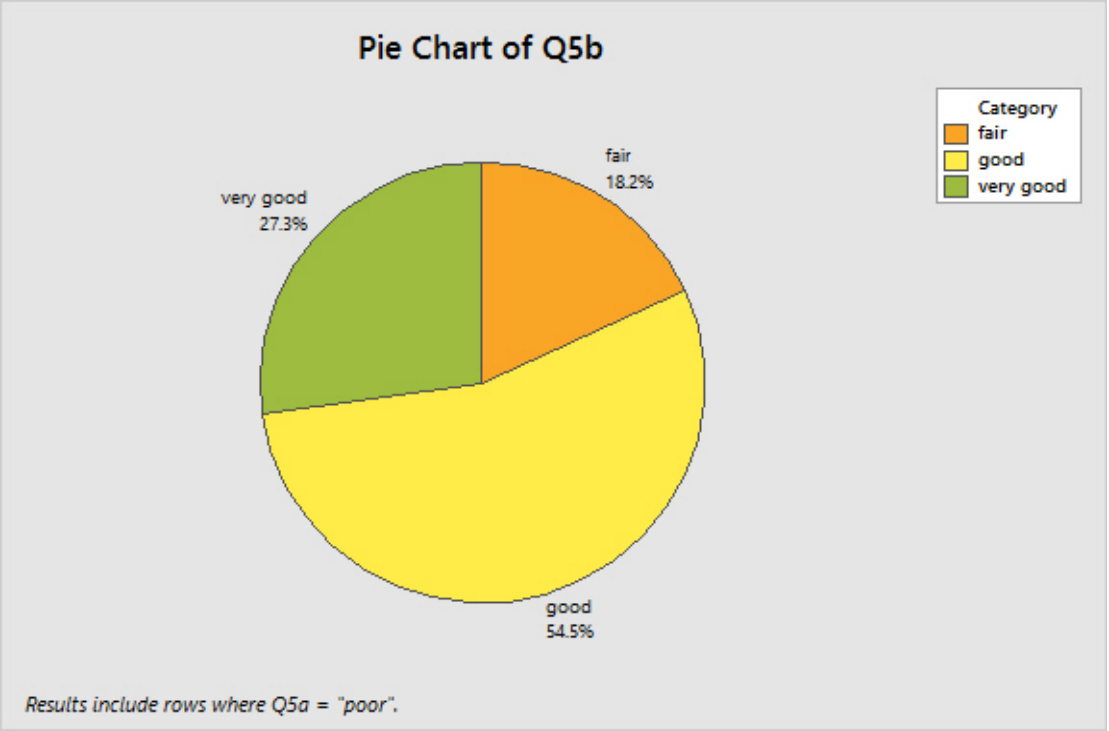


FIGURE 4.5: How students who previous ranked themselves as poor, ranked after

improvement in how the participants graded themselves after using the application. The majority of the participants estimated their knowledge on the subject as good after viewing the animation which is an increase of two on the scale.

Figure 4.5, shows the breakdown of results for a subset of students after the animation. These students had classed themselves as poor before viewing the animation. This breakdown displays an increase in knowledge of the students after viewing the animation.

The goal of this research was to educate and encourage the students of the compiler design module on register allocation via graph colouring. The results in figure 4.5, give some indication of how this application performed.

It is also worth mentioning that all participants enjoyed using the application and would recommend it to a friend not to mention, they found it easy to use. This includes students who do not have an interest in compiler design. This feedback

is important in relation to the application and it's goal. It is hoped that future classes of the compiler design module would enjoy using it as well as finding it easy to operate.

While the participants were using the application some other points that were not covered in the survey became apparent.

One difficulty many of the participants encountered was with the liveness analysis. The equations for the liveness analysis are displayed on the screen for the entire analysis and depict two equations that contain basic set theory operators. The difficulty arose from the participants not knowing that the liveness analysis was performed using sets. Although this distinction is not obviously stated, the word set is mentioned in one of the sentences displayed at the bottom of the screen. It seems that this was not enough for the participants to grasp the concept. On the other hand, these students had not studied the second compiler design module where liveness analysis is taught in depth. Furthermore, the equations in question (discussed in section 2.2.2) do require a lengthy explanation in order to grasp the theory correctly however, as this is discussed in the compiler design module it seems unnecessary to include it in the animation and it would not make for a nice clean graphical animation.

Another challenge faced by some of the participants was the operation of Vivio. A sheet with instructions for Vivio was given to each student at the start of the experiment in the event they were unfamiliar with it but it seems most participants opted not to use this instruction sheet. Including instructions on screen for this application would be excessive for the scale of the application. Likewise, all other Vivio animations do not have this feature and it is assumed the users will embrace the on-line documentation for the instructions of Vivio.

While completing this dissertation time was of the essence and therefore little time was left for experiments. If there had been an abundance of time available more concrete experiments could have been performed. If this study was to be completed again in the future with more time available, some improvements could certainly be made.

Firstly, two surveys should have been completed so the user could not anticipate the questions and their answers to them. One survey would be given before the student used the application and the other would be given after the student had used the application. The second survey could have included some questions on the subject they had just viewed. Additionally, having further segregation between the users and the researcher would perhaps encourage the participants to be more honest as their data would not be directly linked to them. All of these improvements would also reduce the effect of *response bias* mentioned above. The amount of participants for this study was quite small and in order to find a more realistic estimate this study should be done on a larger scale. Migrating this study to be done over the Internet could assist in these modifications.

Chapter 5

Future Work

The goal of this project was to educate students of the compiler design module on register allocation via graph colouring. This application has had some success in achieving this goal, be that as it may, there are endless opportunities to expand on the project.

An option for the user to enter a program could be implemented as an enhancement. Allowing the user to enter a program they understand would have the benefits of the user already knowing the program and focusing on the specific subject. It could also demonstrate to them how this process would work on an infinite set of programs, thus leading to more examples for them to follow. There are many obstacles that would have to be addressed for this to be implemented. These would include the following:

1. A standard format for the inputted program would need to be established in order for the application to correctly parse it.

2. There may need to be a limit on the size of the program due to the amount of available screen space. If the user was to enter a program that was very large

and required a lot of spilling, there may be a situation where the program becomes too large to display on screen or, if scaled to fit on the screen, would be unreadable.

3. Finally the current version of Vivio does not have the capabilities to read text input from the user. This would need to be implemented by the researcher along with the new application. Vivio has had a lot of new releases, so it is possible this feature may be added in a future release.

Another aspect that could be implemented to complement the current application, would be to complete animations for the other stages of the compiler. There are many stages in the compiler pipeline that are not covered in this animation. Adding these to the animation would give the user a more complete picture of the operations involved in compiling code. This could include, but is not limited to:

1. Lexical Analysis
2. Intermediate code generation
3. Code optimisations

Likewise this software could be used in the compiler design module to educate the students on the complete workings of the compiler. If these additions were to be implemented the current application could be used as a starting point to build on.

As mentioned in section 2.5, there are other similar works published that focus on improving Chaitin's [2] algorithm. Other animations could be created, similar to this, to execute different approaches, which could further the students education on the subject of register allocation. These other papers are currently touched upon in the compiler design module and it would be beneficial for the students to see these approaches operational. Using the same sample programs for the different algorithms

would allow the students to easily compare them i.e. where one algorithm needs to spill but the other does not, a point mentioned by Briggs [1].

Even though this project has reached its conclusion there is potential for building on this. Results have shown minor improvements in educating students on this subject and in consequence any additional elements could help improve the range of knowledge of the students.

Chapter 6

Conclusion

The goal of this project was to create an E-Learning animation of register allocation via graph colouring. This goal has been achieved. The animation displays the complete process of register allocation via graph colouring as set out by Chaitin [2] and has been completed in full.

The motivation for this application was to create an animation of register allocation that would assist students of the compiler design module in Trinity College Dublin, to understand this topic. This application is ready for students to use in the module next semester.

The animation was created using Vivio which possess many features to enhance the user's experience. The animation can be completely controlled by the user, including, starting and stopping the animation at any point, controlling the speed of the animation and playing the animation backwards in case something was missed. Vivio animations are created with the main publication medium being the World Wide Web. This animation will be available to students on the Internet meaning they will have access to it at any time.

Experiments were used to assess how advantageous this application was in communicating this subject. Users were asked to use the application and fill in a short survey which included questions on their current stance on the topic and their experience of the application. The results of these experiments show some improvement in the users' perception of their own knowledge of the subject after viewing the animation. Besides this, all the participants not only enjoyed using the application but would recommend it to others with a desire to learn about this subject.

These results should be analysed while taking into consideration the matter of *response bias*. *Response bias* refers to the behaviour a participant in the study may display when answering questions and can be influenced by the intimate nature of the study. Participants may give answers they think the researcher wants rather than giving their true answers. What's more, most results display an increase in the users knowledge, hence, it is acceptable to say there are benefits from using this application. If there was an abundance of time a more thorough study could have been conducted.

Overall it seems apparent that this was a successful and worthwhile project, and was useful in the student's understanding of this subject.

Appendix A

Algorithms

```
input  : Program String
output: Program in a formatted string, an array of program instructions
while currentChar < program.len() do
    currentInstruction =“”;
    while program[currentChar] != “;” && program[currentChar] != “{” &&
    program[currentChar] != “}” do
        currentInstruction += program[currentChar];
        currentChar++;
    end
    currentInstruction += program[currentChar];
    programInstructions.add(currentInstruction);
    if program[currentChar] == “}” then
        stack–;
    end
    for i ← 0 to stack do
        formattedProgram += indent;
    end
    if program[currentChar] == “{” then
        stack++;
    end
    formattedProgram += currentInstruction + “newline”;
    currentInstruction =“”;
    currentChar++;
end
return formattedProgram;
```

Algorithm 1: Parsing the program

Data: update information
Used to reduce repetition in the next algorithm;
increment instruction count in current information;
add current basic block to array of basic blocks;
set previous block number to current basic block;
increment current block in current information;

Algorithm 2: update information

```
input  : Array of program instructions, Structure of current information
output: Array of the Basic blocks of the program
while not at end of Array of program instructions do
    read current;
    if contains '}' then
        if closing a while loop then
            add in corresponding instruction
        end
        update information;
        return
    end
    if contains 'return' then
        update information;
        add instruction to current basic block and increment count;
        increment current block in current information;
    end
    if contains 'if' then
        update information;
        add instruction to current basic block and increment count;
        add instruction to expression stack in current information;
        current information ← findBasicBlocks(instructions, current
        information);
    end
    if contains 'while' then
        update information;
        add instruction to expression stack in current information;
        current information ← findBasicBlocks(instructions, current
        information);
    end
    if contains 'else' then
        update information;
        add instruction to expression stack in current information;
        current information ← findBasicBlocks(instructions, current
        information);
    end
    if contains none of the above then
        add instruction to current basic block and increment count;
        increment instruction count in current information;
    end
    if reached end of conditional block then
        create a new basic block;
        connect any unconnected if blocks;
    end
end
add current basic block to array of basic blocks;
```

Algorithm 3: Basic Block creation: findBasicBlocks

input :
output:

```
int changed = 1;
while changed is 1 do
    changed = 0;
    for bb ← nunmberOfBB to 0 do
        find live out of bb;
        globalLivenessAnalysis[bb].liveout = liveOut;
        oldLiveIn = globalLivenessAnalysis[bb].livein ;
        find livein of bb ;
        globalLivenessAnalysis[bb].livein = livein ;
        if livein ≠ oldLiveIn then
            changed = 1;
        end
    end
end
```

Algorithm 4: Iterative Global Flow Analysis

input : Stack size
output: node to spill

```
for n ← 0 to N do
    if still in the graph then
        real heuristic ← cost[n]/degree[n];
        if heuristic < currentLowest && n ∉ spilllist && n ∉ stack then
            currentLowest ← heuristic;
            nodeToSpill ← n;
        end
    end
end
add nodeToSpill to spill list;
return nodeToSpill;
```

Algorithm 5: Find node to spill

Appendix B

Experiment Documentation

TRINITY COLLEGE DUBLIN
INFORMED CONSENT FORM

LEAD RESEARCHERS: Caoimhe O'Regan

BACKGROUND OF RESEARCH: The project aims to develop a software tool for visualising an algorithm for register allocation via graph colouring. It is hoped, if this project is successful, that the application will be used to help future years in understanding the process of graph colouring. The purpose of this research is to help determine the success of the application.

PROCEDURES OF THIS STUDY: The participant is asked to use the application for a period of 20-30 minutes and then complete the survey which should take approximately 2 minutes.

PUBLICATION: Survey data will be anonymised and only trends will be reported in the dissertation. All survey data will be held in accordance with the Data Protection Act

Individual results will be aggregated anonymously and research reported on aggregate results.

DECLARATION:

- I am 18 years or older and am competent to provide consent.
- I have read, or had read to me, a document providing information about this research and this consent form. I have had the opportunity to ask questions and all my questions have been answered to my satisfaction and understand the description of the research that is being provided to me.
- I agree that my data is used for scientific purposes and I have no objection that my data is published in scientific publications in a way that does not reveal my identity.
- I understand that if I make illicit activities known, these will be reported to appropriate authorities.
- I freely and voluntarily agree to be part of this research study, though without prejudice to my legal and ethical rights.
- I understand that I may refuse to answer any question and that I may withdraw at any time without penalty.
- I understand that my participation is fully anonymous and that no personal details about me will be recorded.
- I understand that if I or anyone in my family has a history of epilepsy then I am proceeding at my own risk.
- I have received a copy of this agreement.

PARTICIPANT'S NAME:

PARTICIPANT'S SIGNATURE:

Date:

Statement of investigator's responsibility: I have explained the nature and purpose of this research study, the procedures to be undertaken and any risks that may be involved. I have offered to answer any questions and fully answered such questions. I believe that the participant understands my explanation and has freely given informed consent.

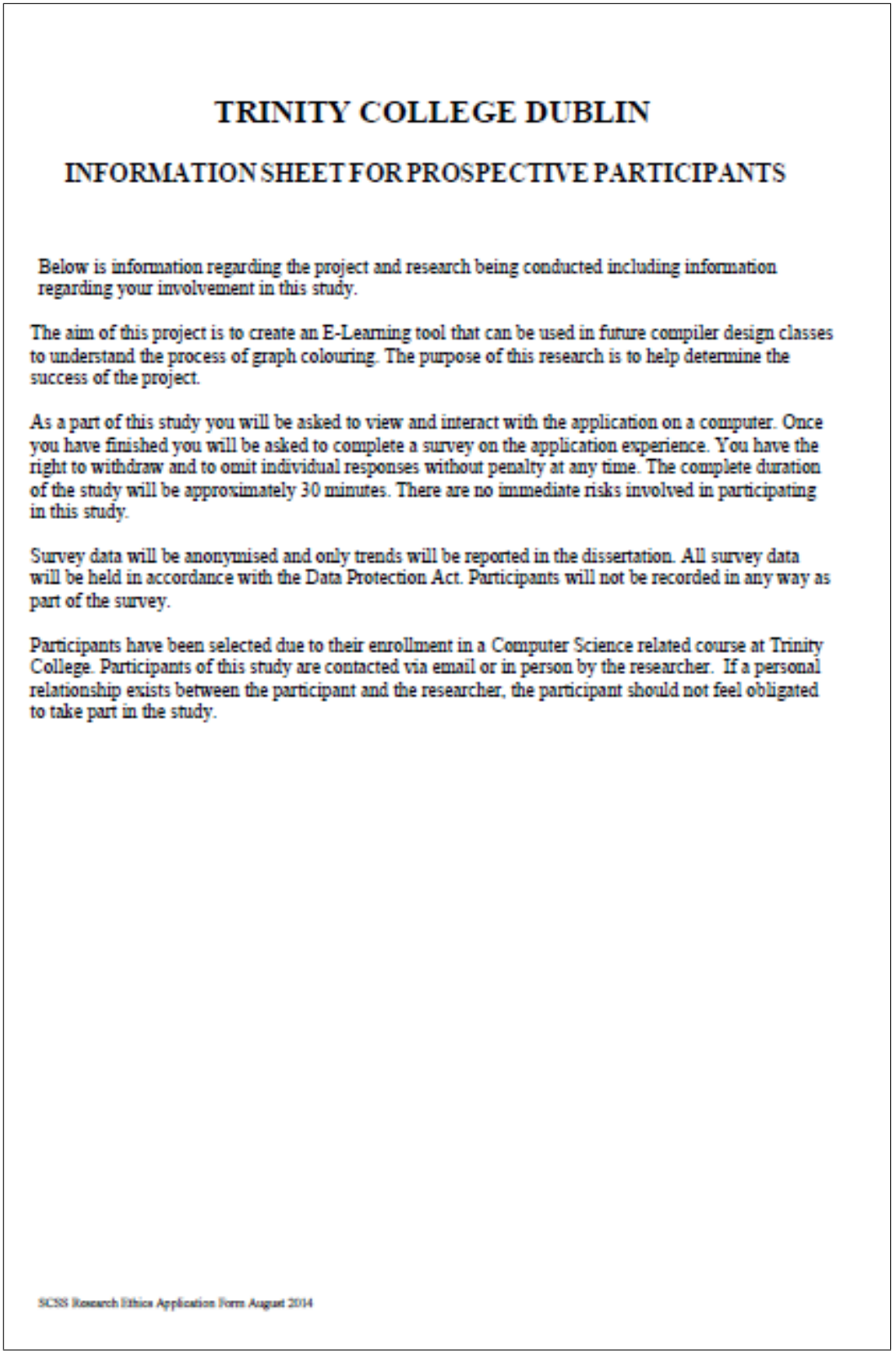
RESEARCHERS CONTACT DETAILS: Email: oreganc2@tcd.ie **Phone: 083 3534662**

INVESTIGATOR'S SIGNATURE:

Date:

SCSS Research Ethics Application Form August 2014

FIGURE B.1: Consent form



TRINITY COLLEGE DUBLIN

INFORMATION SHEET FOR PROSPECTIVE PARTICIPANTS

Below is information regarding the project and research being conducted including information regarding your involvement in this study.

The aim of this project is to create an E-Learning tool that can be used in future compiler design classes to understand the process of graph colouring. The purpose of this research is to help determine the success of the project.

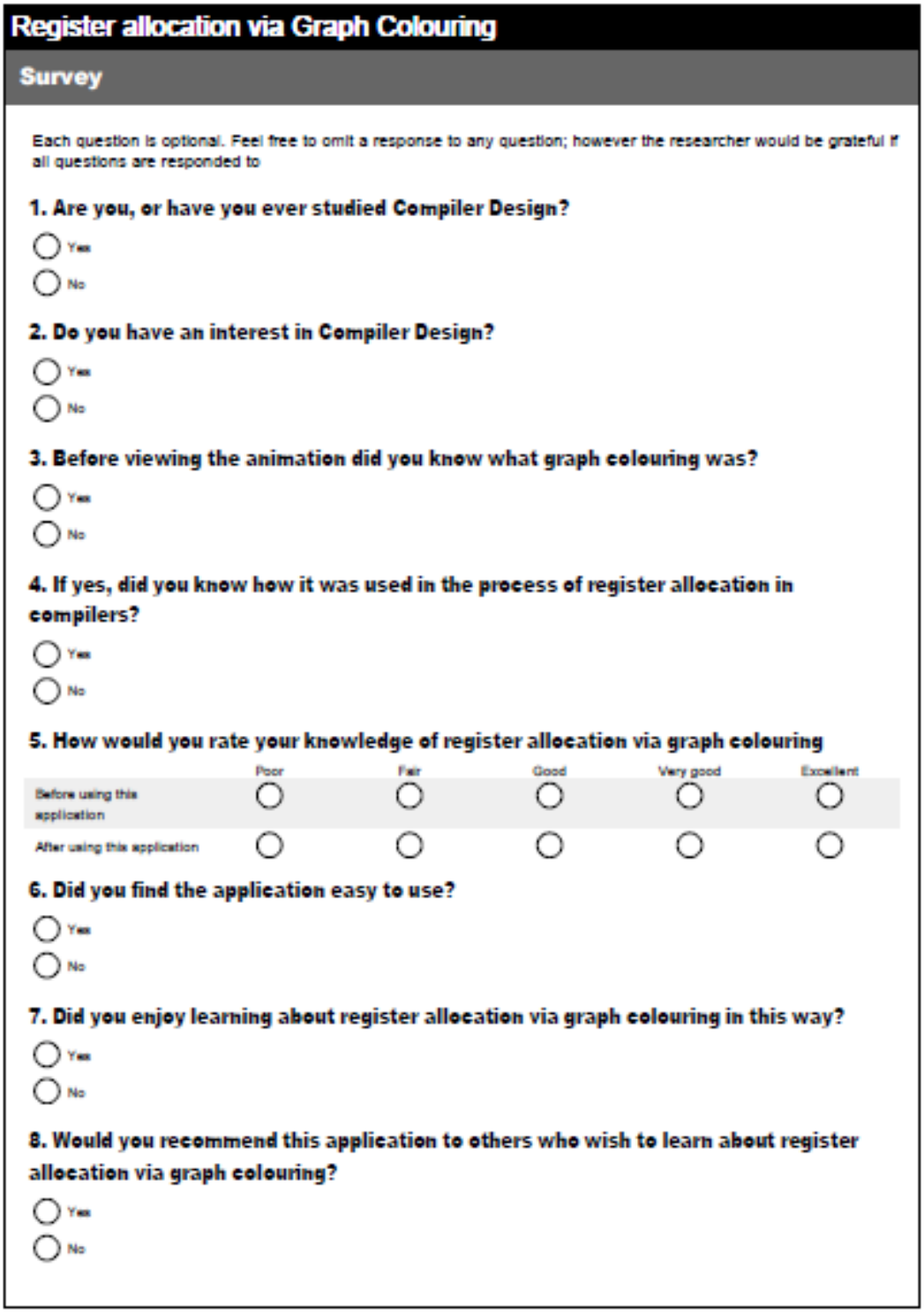
As a part of this study you will be asked to view and interact with the application on a computer. Once you have finished you will be asked to complete a survey on the application experience. You have the right to withdraw and to omit individual responses without penalty at any time. The complete duration of the study will be approximately 30 minutes. There are no immediate risks involved in participating in this study.

Survey data will be anonymised and only trends will be reported in the dissertation. All survey data will be held in accordance with the Data Protection Act. Participants will not be recorded in any way as part of the survey.

Participants have been selected due to their enrollment in a Computer Science related course at Trinity College. Participants of this study are contacted via email or in person by the researcher. If a personal relationship exists between the participant and the researcher, the participant should not feel obligated to take part in the study.

SCSS Research Ethics Application Form August 2014

FIGURE B.2: Information sheet



Register allocation via Graph Colouring

Survey

Each question is optional. Feel free to omit a response to any question; however the researcher would be grateful if all questions are responded to

1. Are you, or have you ever studied Compiler Design?

- ◯ Yes
- ◯ No

2. Do you have an interest in Compiler Design?

- ◯ Yes
- ◯ No

3. Before viewing the animation did you know what graph colouring was?

- ◯ Yes
- ◯ No

4. If yes, did you know how it was used in the process of register allocation in compilers?

- ◯ Yes
- ◯ No

5. How would you rate your knowledge of register allocation via graph colouring

	Poor	Fair	Good	Very good	Excellent
Before using this application	◯	◯	◯	◯	◯
After using this application	◯	◯	◯	◯	◯

6. Did you find the application easy to use?

- ◯ Yes
- ◯ No

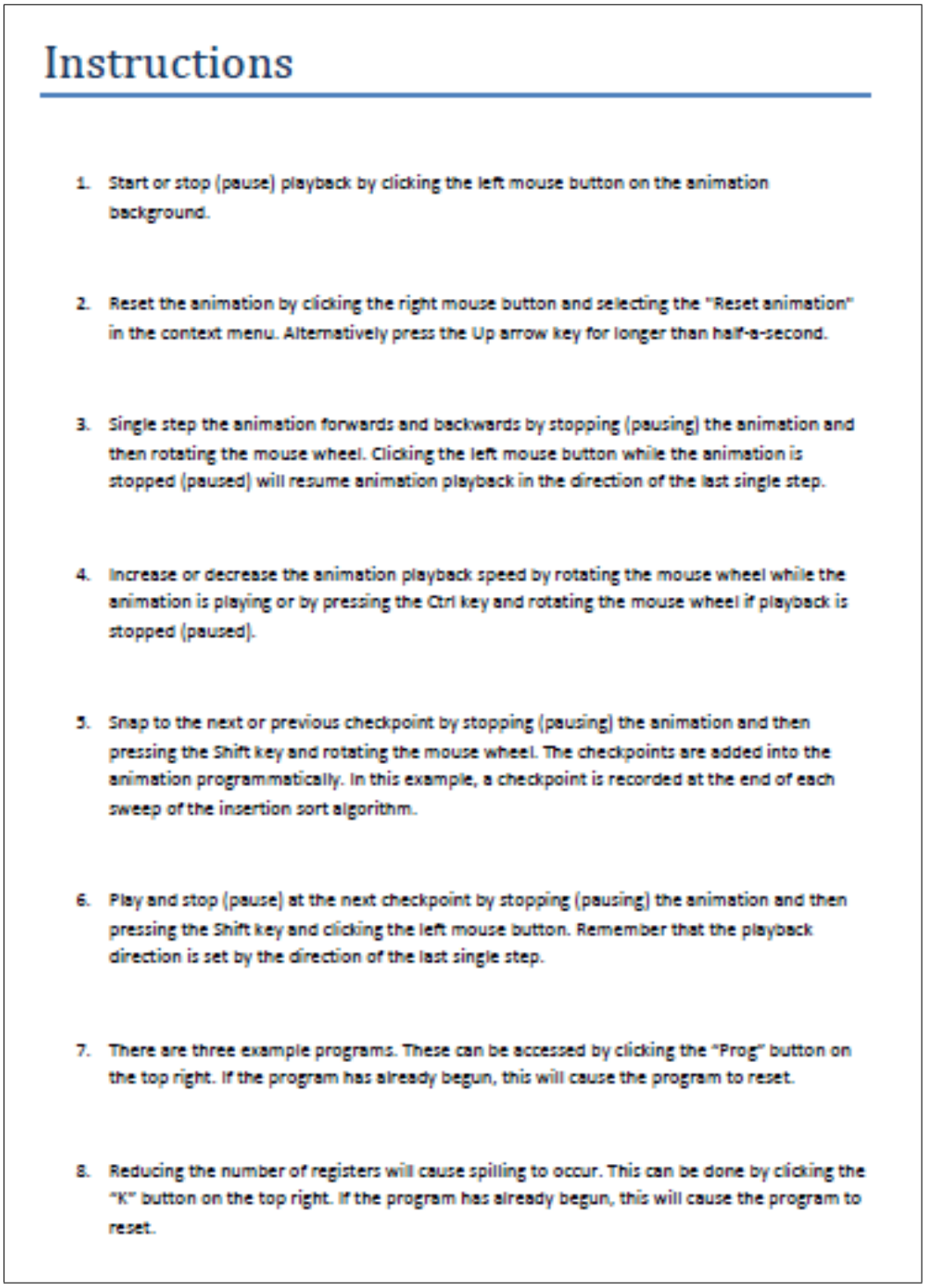
7. Did you enjoy learning about register allocation via graph colouring in this way?

- ◯ Yes
- ◯ No

8. Would you recommend this application to others who wish to learn about register allocation via graph colouring?

- ◯ Yes
- ◯ No

FIGURE B.3: Survey given to the participants



Instructions

1. Start or stop (pause) playback by clicking the left mouse button on the animation background.

2. Reset the animation by clicking the right mouse button and selecting the "Reset animation" in the context menu. Alternatively press the Up arrow key for longer than half-a-second.

3. Single step the animation forwards and backwards by stopping (pausing) the animation and then rotating the mouse wheel. Clicking the left mouse button while the animation is stopped (paused) will resume animation playback in the direction of the last single step.

4. Increase or decrease the animation playback speed by rotating the mouse wheel while the animation is playing or by pressing the Ctrl key and rotating the mouse wheel if playback is stopped (paused).

5. Snap to the next or previous checkpoint by stopping (pausing) the animation and then pressing the Shift key and rotating the mouse wheel. The checkpoints are added into the animation programmatically. In this example, a checkpoint is recorded at the end of each sweep of the insertion sort algorithm.

6. Play and stop (pause) at the next checkpoint by stopping (pausing) the animation and then pressing the Shift key and clicking the left mouse button. Remember that the playback direction is set by the direction of the last single step.

7. There are three example programs. These can be accessed by clicking the "Prog" button on the top right. If the program has already begun, this will cause the program to reset.

8. Reducing the number of registers will cause spilling to occur. This can be done by clicking the "K" button on the top right. If the program has already begun, this will cause the program to reset.

FIGURE B.4: Vivio instructions given to participants

Bibliography

[1] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. *SIGPLAN Not.*, 39(4):283–294, April 2004. ISSN 0362-1340. doi: 10.1145/989393.989424. URL http://doi.acm.org/10.1145/989393.989424.

[2] Gregory Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Not.*, 39(4):66–74, April 2004. ISSN 0362-1340. doi: 10.1145/989393.989403. URL http://doi.acm.org/10.1145/989393.989403.

[3] J. Jones. Vivio - a system for creating interactive reversible e-learning animations for the www. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 131–133, Sept 2004. doi: 10.1109/VLHCC.2004.63.

[4] Trinity College Dublin School of Computer Science. Computer science. URL https://www.scss.tcd.ie/undergraduate/computer-science/.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

[6] Torben Ægidius Mogensen. *Introduction to Compiler Design*. Undergraduate Topics in Computer Science. Springer, 2011. ISBN 978-0-85729-828-7. doi: 10.1007/978-0-85729-829-4. URL http://dx.doi.org/10.1007/978-0-85729-829-4.

[7] Oxford University Press. Oxford dictionaries. URL http://www.oxforddictionaries.com/.

[8] Michael D. Byrne, Richard Catrambone, and John T. Stasko. Evaluating animations as student aids in learning computer algorithms. *Computers & Education*, 33(4):253 – 278, 1999. ISSN 0360-1315. doi: http://dx.doi.org/10.1016/S0360-1315(99)00023-8. URL http://www.sciencedirect.com/science/article/pii/S0360131599000238.

[9] Gary B. Mayton. Learning dynamic processes from animated visuals in microcomputer-based instruction. page 1, 1991. URL http://eric.ed.gov/?id=ED334999.

[10] Michael R. Birch, Christopher M. Boroni, Frances W. Goosey, Samuel D. Patton, David K. Poole, Craig M. Pratt, and Rockford J. Ross. Dynalab: A dynamic computer science laboratory infrastructure featuring program animation. *SIGCSE Bull.*, 27(1):29–33, March 1995. ISSN 0097-8418. doi: 10.1145/199691.199706. URL http://doi.acm.org/10.1145/199691.199706.

[11] Dr. Jeremy Jones. *Vivio Reference*. Trinity College Dublin.

[12] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47 – 57, 1981. ISSN 0096-0551. doi: http://dx.doi.org/10.1016/0096-0551(81)90048-5. URL http://www.sciencedirect.com/science/article/pii/0096055181900485.

[13] Fred Chow and John Hennessy. Register allocation by priority-based coloring. *SIGPLAN Not.*, 39(4):91–103, April 2004. ISSN 0362-1340. doi: 10.1145/989393.989406. URL http://doi.acm.org/10.1145/989393.989406.

[14] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, July 1983. ISSN 0004-5411. doi: 10.1145/2402.322385. URL http://doi.acm.org/10.1145/2402.322385.

[15] Frederick Chi-Tak Chow. *A Portable Machine-independent Global Optimizer–design and Measurements*. PhD thesis, Stanford, CA, USA, 1984. AAI8408268.

[16] Douglas Cunningham. *Experimental design from user studies to psychophysics*. CRC Press, Boca Raton, FL, 2012. ISBN 978-1-56881-468-1.