Compiling for Reduced Register Pressure

by Jarrod Richardson

Supervisor: Prof. David Gregg

School of Computer Science and Statistics

Department of Electronic and Electrical Engineering

M.A.I in Electronic & Computer Engineering

Trinity College Dublin

Declaration

I, Jarrod Richardson, 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:

Dated:

Summary

One of the best-known compiler optimizations is instruction scheduling, which re-orders instructions to better exploit pipelining and instruction-level parallelism. When the compiler re-orders instructions in this way, the result is often an increase in the number of local variables and values that are simultaneously live. This also causes the number of live local values to exceed the number of available registers and the result of this is that some live values have to be spilled to memory.

Despite all the existing compiler work on instruction scheduling, it is not clear that it is actually solving the right problem for popular out-of-order processors that have few architected registers and do instruction scheduling in hardware. Instead, there may be benefits in the compiler trying to re-order instructions to reduce the number of simultaneously live values. This is otherwise known as reducing the register pressure.

For this project, an additional compiler optimization pass was developed to reduce register pressure for a basic block of code. The pass was written for LLVM, a compiler framework library, to test and determine whether reducing register pressure is a viable solution for processors with few architected registers. The paper explores a number of examples that have reduced register pressure solutions and tries to discern where reducing register pressure can improve the overall optimization and efficiency.

In addition to this, the use of randomly generated data dependency graphs were used to comprehensively test the instruction scheduler designed for the optimization pass. These were tested thousands of times to find the average reduced register pressure gained when using the instruction scheduler. It also highlights a number of variables that have an effect on these figures, and how

well randomly generated data dependency graphs reflect the real world.

Acknowledgements

I am using this opportunity to express my gratitude to everyone who supported me throughout the course of this MAI project. I am thankful for their aspiring guidance, invaluably constructive criticism and friendly advice during the project work. I am sincerely grateful to them for sharing their truthful and illuminating views on a number of issues related to the project.

I express my thanks to Prof. David Gregg and my progenitors, Mark and Carol, for their support and guidance.

Contents

List of Figures

List of Tables

Terms of Reference

Term	Acronym	Description
Central Processing Unit	CPU	The electronic circuitry within a computer
Vertex/Vertices	V	A point where two or more curves, lines, or edges meet.
Edges	E	A particular type of line segment joining two vertices.
Low level Virtual Machine	LLVM	A collection of modular and reusable compiler and tool-chain technologies used to develop compiler front ends and back ends
Data Dependence	DD	A situation in which an instruction is dependent on a result from a sequentially previous instruction before it can complete its execution.
Data dependency graph	DDG	A graph representing data dependence.
Chain	C	A sequence of adjacent edges.
Chain colouring		Assigning a register to a chain
Assembly		Low level programming language .
Operation code	Opcode	The portion of a machine language instruction that specifies the operation to be performed
Depth-first search		An algorithm for traversing or searching tree or graph data structures
Immediate representation	IR	The data structure or code used internally by a compiler or virtual machine to represent source code.

Table 1: A List of Terms of Reference I

Term	Acronym	Description
Advanced RISC machine	ARM	A processor architecture based on a 32-bit reduced instruction set (RISC) computer.
Arithmetic Logical Unit	ALU	A unit in a computer which carries out arithmetic and logical operations
Throughput		The number of instructions that can be executed in a unit of time
Jumps		Machine level branch instructions.
Random Access Memory	RAM	A form of computer data storage which stores frequently used program instructions.
Read Only Memory	ROM	A form of computer data storage used to store the start-up instructions for a computer.

Table 2: A List of Terms of Reference II

Chapter 1

Introduction

The objective of this paper is to discuss how the design and implementation of a compiler optimizer attempts to reduce register pressure. This is achieved by re-ordering and re-naming instructions within an instruction schedule or a basic block of instructions. This is done to account for the available registers to try and prevent memory spills, as memory spills inhibit the overall performance or throughput in terms of execution time for a basic block of instructions. The incentive for reducing register pressure, is to determine if there is a benefit in the compiler re-ordering the instruction schedule, as it is specifically useful for processors that have few registers and will benefit the most from reducing the register pressure.

Chapter 2

Literature Review

2.1 Central Processing Units

The Central Processing Units (CPUs) within computers are the central electronic circuitry that carry out instructions from a computer program that performs things like basic arithmetic, logical, control and input/output operations specified by these instructions [5].

Computer architecture has advanced significantly since the early designs in the 1950s [11]. Back then, each computer design was unique as there were no general purpose computers. These computer designs were often designed to handle a specific problem and would share no or few similarities between other computer designs. As a result of this the software built for each of these computers was different and could not be shared between other computer types. Today, general purpose computers number in the billions and they are capable of executing an array of useful and widely applicable instructions that can handle a multitude of different computer programs that solve any number of problems.

The Central Processing Unit does not operate in solitude within the computer. There are a number of other components that a CPU can communicate with. These components help make up the rest of the computer in addition to the CPU and some of these can be seen in Table 2.1.

Component	Description
Primary Memory	Responsible for storing CPU program instructions and any data actively operated on by the computer.
Auxiliary Memory	Responsible for keeping any data when the computer is powered down and also miscellaneous data transferred from Primary.
Input/Output Interfaces	Responsible for connecting and communicating the computer to the outside world, where various data can be sent/received.

Table 2.1: Other Central Processing Unit Components

2.2 Inside the Central Processing Unit

2.2.1 The Registers

Processor registers are memory locations available to a computer's central processing unit. These are usually small and fast storage devices that are contained within the CPU [5]. Some but not all registers can have specific hardware functions and may be read-only or write-only. Registers are of a set size of N Bits, which depend on the computer's architecture or the specific operation of the register. In a CPU the number of registers is finite and data that does not fit within the finite amount of registers needs to be stored in some other storage location such as primary memory.

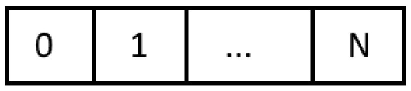


Figure 2.1: An N Bit Register

2.2.2 The Register File

The register file is an array of processor registers within the central processing unit [5]. These registers are available to the programmer and are used to stage data between memory and the functional units on the CPU. In more complicated CPUs, a technique called register renaming is used to eliminate false data dependencies that arise from the reuse of processor registers by successive instructions that don't have any real data dependence between them [10].

2.2.3 The Arithmetic Logic Unit

The arithmetic logic unit (ALU) is a type of functional unit that resides within the processor. It is a digital circuit that performs mathematic arithmetic and bitwise logic operations for a given set of inputs [5]. These inputs to the arithmetic logic unit are seen in Table 2.2, where a brief description of each is given. A visual representation of the arithmetic logic unit is shown in Figure 2.2. The central processing unit sends off all operations associated with the arithmetic logic unit to the ALU unit. The other CPU operations are not handled by the arithmetic logic unit. The ALU often has a set of internal registers that the operands and results are stored in while the ALU performs its operations.

Inputs	Description
Operands	A number of operands supplied by the register file.
Opcode	A code which indicates which operations the ALU is to complete.
Status	Information from the previous operations of the ALU.

Table 2.2: The Inputs to the Arithmetic Logic Unit

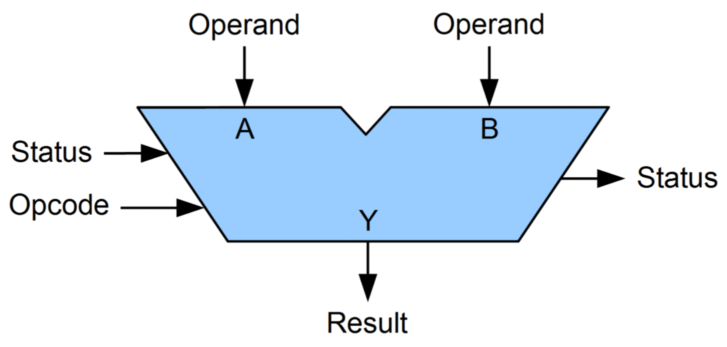


Figure 2.2: Arithmetic Logic Unit

The outputs from the arithmetic logic unit are seen in Table 2.3 where a brief description is also given for each one of them. The result of an operation performed by the ALU is deterministic in nature as there is no randomness involved with the output for a given set of inputs. A system that is deterministic will always produce the same result for the same set of inputs. The result and the status information are both deterministic. The outputs of the ALU are visible to the programmer via the register file. The result can be either transferred to any of the operands for the next operation of the ALU or back into the register file to be saved in a register. The status information is also transferred back into the ALU for the next operation.

Outputs	Description
Result	The output that results from the combination of inputs.
Status	Information of the current operation of the ALU.

Table 2.3: The Outputs of the Arithmetic Logic Unit

2.3 The Operation of the Central Processing Unit

The fundamental operation of the CPU is to execute a sequence of stored instructions that is commonly referred to as a program. The instructions to be executed are kept in computer memory which are then transferred to the central processing unit for execution. There is common architecture in most CPUs that follow a set of steps that break up executing a single instruction into multiple steps, each handled by a different part of the CPU. These steps include: fetching the next instruction; decoding the instruction; and executing the instruction. This type of architecture is collectively known as an instruction cycle pipeline [5]. Pipelining instructions implement a form of parallelism that is said to speed up the execution of a single processor.

After the execution of an instruction, the entire process is repeated using the next instruction cycle, normally fetching the next-in-sequence instruction because of the incremented value in the program counter. If a jump instruction was executed, the program counter will be modified to contain the address of the instruction that had been jumped to and the program execution will continue as normal.

Some instructions manipulate the program counter rather than producing result data directly. These instructions are referred to as “jumps” and facilitate program behaviour like loops and conditional program execution and calling out to a function via a conditional jump.

2.3.1 Processor Instructions

An instruction is a single operation of a processor defined by the processor instruction set [8]. This set of instructions is dependent on the computer architecture itself. There is often a common set of instructions implemented by almost every central processing unit. Some common instructions are those that involve the arithmetic logic unit or sending data to and from the memory units connected to the CPU. On traditional architectures, an instruction is a sequence of bits (often referred to as machine code) that includes an opcode, along with any operands that the opcode requires to execute. The instruction bits are usually divided into a set number of bits for each component of the instruction, for example see Figure 2.3 where the opcode on this particular system is the first 6 bits, with the total number of bits for the entire instruction at 33 bits. The remaining bits are divided into sections that would be specific to a particular opcode, in this case the ”Destination” is 10 bits, the ”OperandA” is 9 bits and the ”OperandB” is 8 bits.

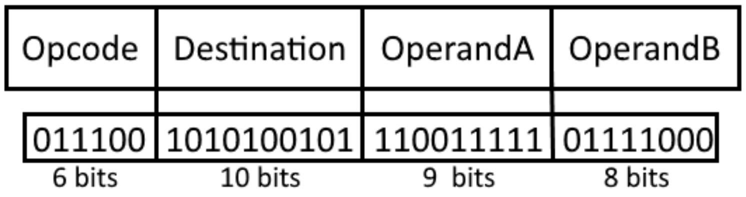


Figure 2.3: An Example of the division of bits in an Instruction

The binary bits used in instructions are for the most part only used and seen by the central processing unit. Programmers instead use human readable code that a they can quickly read and use to program the central processing unit. An example of human readable code is the ARM assembly language [1]. Shown in Figure 2.4 is a simple addition instruction that takes two operands and stores the result in a register located in the register file.

```
add r1, r0, 2
```

Figure 2.4: ARM Assembly Addition Instruction

An English translation of this instruction is as follows: "Add two to the contents of register zero and store the result in register one". To read this instruction, start first with prefixed word "add", this correspond to the opcode that adds two numbers together. The next word is "r1", this corresponds to register one within the register file. The location of this register in the instruction indicates that it is the destination register of this operation. This means that the result will be stored in register one. Commas are used to break up the operands of an instruction. The following two operands are "r0" and "2", which are register zero and constant two respectively. One could also represent the instruction mathematically as

$$r1 = r0 + 2$$

Human readable code is converted into machine code with the use of a assembler/compiler program. Often the Human readable code is simplified such that is it easier to manage/program and large tedious tasks are shortened, this requires the assembler/compiler program to convert and expand the code into all of the required machine code.

2.3.2 Instruction cycle pipelines

Instruction pipelining is a technique that implements a form of parallelism called instruction-level parallelism within a single processor [5]. It therefore allows faster CPU throughput (i.e. the number of instructions that can be executed in a unit of time) than would otherwise be possible at a given clock rate.

The instruction cycle is split into a series called a pipeline. Rather than processing each instruction sequentially, each instruction is split up into a sequence of dependent steps so different steps can be executed in parallel and instructions can be processed concurrently.

The first step is always to fetch the instruction from memory; the final step is usually writing the results of the instruction to the processor registers or to memory. Pipelining seeks to let the processor work on as many instructions as there are dependent steps, just as an assembly line builds many vehicles at once, rather than waiting until one vehicle has passed through the line before admitting the next one. Just as the goal of the assembly line is to keep each assembler productive at all times, pipelining seeks to keep every portion of the processor busy processing an instruction. Pipelining lets the computer's cycle time be the time of the slowest step, and ideally lets one instruction complete in every cycle.

2.4 Primary and Auxiliary Memory

Primary memory is often referred to as main memory. Main memory is the only memory directly accessible to the CPU. The CPU continuously reads instructions stored there and executes them as required. Any data actively operated on is also stored there in a uniform manner [13].

Main memory is directly or indirectly connected to the central processing unit via a memory bus. It consists of two separate buses namely an address bus and a data bus. The CPU firstly sends a number through an address bus that indicates the desired location of the data. It then reads or writes the data in the memory cells using the data bus.

Other memory is auxiliary memory which is used to store large amounts of data that is not frequently used by the CPU. It is also used for saving data when the computer powers down as main memory is not preserved without power [13].

The access speed of primary and auxiliary memory is much slower than that of memory that sits directly within the central processing unit. Primary memory access is faster than auxiliary memory. This difference in speed between CPU and other 2 memory types is relevant, because the CPU has to enter a wait state after requesting data from either one of these two memory types. The CPU has to wait several cycles before it can continue executing instructions if the last instruction was to fetch data from memory.

2.5 Instruction Scheduling

Instruction scheduling is a compiler optimization used to improve instruction-level parallelism, which improves performance on machines with instruction pipelines. Without changing the meaning of the code, it tries to avoid pipeline stalls by rearranging the order of instructions and avoid illegal or semantically ambiguous operations.

The problem facing an instruction scheduler is to reorder machine-code instructions to minimize the total number of cycles required to execute a particular instruction sequence. Unfortunately, sequential code executing on a pipelined processor inherently contains dependencies between some instructions. Any transformations performed during instruction scheduling must preserve these dependencies in order to maintain the logic of the code being scheduled.

As well as this, instruction schedulers often have a secondary goal of minimizing register lifetimes or at least not extending them unnecessarily if they are not in use. This is usually a conflicting objective in practice, because limiting the number of live registers introduces false dependencies [5].

2.5.1 Register Spilling

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This introduces data and control hazards. Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on non-pipelined machines [6].

The operation of moving a variable from a register to memory is called spilling, while the reverse operation of moving a variable from memory to a register is called filling. A variable has a much slower processing speed when spilled to memory compared to a variable in a register [14].

Register pressure measures the availability of free registers at each point during the program execution. When a large number of the available registers are in use the register pressure is said to be high. One way to reduced the register pressure is to increase the number of registers in an architecture, however, this increases the cost [12].

2.6 Graph Theory

Graph theory is the study of graphs [15], which are structures of mathematical models that have certain relations between various objects. A graph is made up of a set of vertices or objects, these vertices are connected to each other by edges. These edges represent the relationships between these vertices. A vertex (singular) or vertices (plural) is the term used to describe an object. An edge is the term used to describe the relationship between two objects. From this point onwards all objects will be referred to as vertices and all object relationships will be referred to as edges. An example of a graph is shown in Figure 2.5. In this particular graph there is three vertices (Object 1, Object 2, Object 3) and three edges. This graph can be mathematically expressed as

$$G = (V, E)$$

where G is the graph, V is the set of vertices and E is the set of edges. For this example, $V = \{v_1, v_2, v_3\}$ and $E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3)\}$.

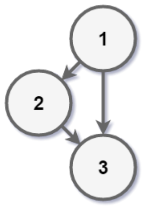


Figure 2.5: A Graph

In this paper, the graphs that are mentioned are all directed graphs. A directed graph is a graph in which the edges have some orientation or direction associated with them.

2.6.1 Graph Colouring

In graph theory, graph colouring is a type of labeling in which vertices of a graph are subject to certain a constraint which dictate their ”colour” [15]. The constraint is that the vertices are coloured such that no two adjacent vertices share the same colour. An example of this is shown in Figure 2.6, it can be seen that no two vertices share the same colour and are adjacent to one another, while there are two vertices that are coloured blue they are not adjacent so they can be coloured the same.

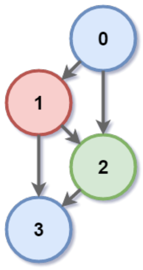


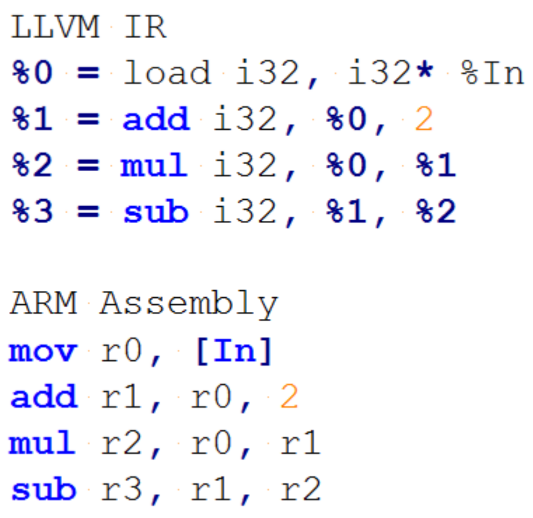
Figure 2.6: A Coloured Graph

Chapter 3

Methodology

3.1 Instruction Schedule

An Instruction Schedule is a sequence of instructions that perform some operation on a central processing unit. The sequence is executed by the processor sequentially in the order that it is given the instructions. An example of a simple instruction schedule in LLVM Intermediate Representation (IR) and ARM Assembly is shown in Figure 3.1. See [1, 9] for details about the language references for both these representations.



```
LLVM IR
%0 = load i32, i32* %In
%1 = add i32, %0, 2
%2 = mul i32, %0, %1
%3 = sub i32, %1, %2

ARM Assembly
mov r0, [In]
add r1, r0, 2
mul r2, r0, r1
sub r3, r1, r2
```

Figure 3.1: An Instruction Schedule in LLVM IR and ARM Assembly

The example in Figure 3.1 shows the same logical set of instructions implemented similarly in the two different human readable representation, first in LLVM IR and then in ARM Assembly. These representations can be read by a compiler and turned into machine code that the processor can understand in order to execute them.

3.2 Data Dependence

Data Dependence, with regards to an instruction schedule, is a situation in which an instruction refers to the data of a preceding instruction. This means that an instruction with a data dependence cannot be executed before the one or more instructions that it depends upon. Some of the instructions listed in Figure 3.1 have a data dependence on other instructions. It can be seen that each instruction (except the first, and in each representation) depends on the previous instruction. The opcode and syntax is of little importance when looking at the data dependence. A simple graph can be created using the representation in Figure 3.1, where only the name and data dependence is preserved. This can be seen in Figure 3.2.

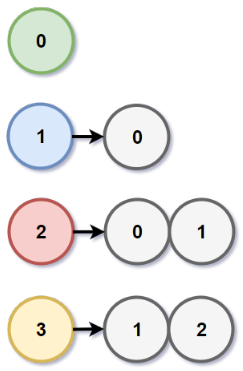


Figure 3.2: An Instruction Schedule Name and Dependence Representation

The coloured circular bubbles on the left, represent the destination of each operation from the instruction schedule. Each row of bubbles corresponds to an instruction, with the arrow indicating the dependence of the left bubble on the right bubbles. The uncoloured circular bubbles on the right of the arrow, represent the operands of the instruction. Throughout this paper these name and data dependence representations will be used instead of an instruction schedule.

3.3 Data Dependence Graph

A data dependence graph (DDG) is a directed graph

$$G = (V, E)$$

where V is the set of vertices that represent the data and E is the set of edges that represent true data dependence between these vertices. A vertex can be connected with multiple incoming edges, meaning the vertex has multiple dependencies on other vertices. A vertex can also be connected with multiple outgoing edges, this means the vertex is depended upon by multiple other vertices.

An example of a data dependence graph is shown in Figure 3.3. This is the data dependence graph of the previous representation in Figure 3.2. In this example, the vertices, V, are represented by the circular bubbles with the vertex index number inside, while the edges, E, are represented by the arrows between the bubbles indicating an edge between the two vertices.
V is enumerated as follows

$$V = (V_0, V_1, V_2, V_3)$$

E is enumerated as follows

$$E = (V_0, V_1), (V_0, V_2),$$

$$(V_1, V_2), (V_1, V_3), (V_2, V_3)$$

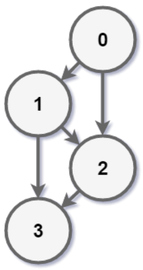


Figure 3.3: A Data Dependence Graph

The importance of making use of a data dependence graph, with regards to reducing the register pressure, is instructions often have data dependence between other registers. The data dependence therefore can be represented in a data dependence graph where the true data dependence is mapped onto the graph. Since re-ordering the instructions doesn't change the data dependence graph, as re-ordering doesn't change any data dependence, this is a useful tool to help model a solution to reduce register pressure.

3.4 Chains

A chain is a sequence of adjacent edges

$$C = (v_1, v_2), (v_2, v_3), \ldots, (v_{k-2}, v_{k-1}), (v_{k-1}, v_k)$$

in a data dependence graph. Chains are formed such that every edge in the data dependence graph is contained within exactly one chain.

An algorithm is needed to traverse a data dependence graph and associate each edge with exactly one chain. The aim is to create the least amount of chains necessary to fully traverse the data dependence graph i.e. the minimum amount.

3.4.1 Minimum Chain Algorithm

In order to attain the minimum number of chains the process requires, a simple depth-first search algorithm can be used. An edge is marked when it is assigned to a chain. A vertex with no outgoing edges or with all of it's outgoing edges marked, is considered the end of a chain. A vertex with no incoming edges or with all of it's incoming edges marked, is considered to be the start of a chain.
The algorithm starts by picking a vertex that is legible to be the start of a chain. It then proceeds to travel the data dependence graph by following one of the outgoing edges, marking the edges as it propagates, until it reaches a vertex that is considered the end of a chain. This process is repeated until every edge is marked and is within exactly one chain.

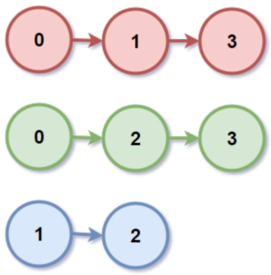


Figure 3.4: A Set of Chains

The chains generated for the data dependency graph in Figure 3.3 are shown in Figure 3.4. There are three chains produced by the minimum chain algorithm, each chain is coloured with a different colour to differentiate between them. The dependency of the chains flows from left to right in the figure. The chains can be formally written as

$$C = [(V_0, V_1), (V_1, V_3)], [(V_0, V_2), (V_2, V_3)], [(V_1, V_2)]$$

3.4.2 Picking Chains

It is important to note that choosing which vertex to start on and choosing which outgoing edge to follow, when a vertex has multiple outgoing edges, will impact the chains that are created. A set of rules needs to be created such that when the algorithm is run for a particular data dependence graph, it either generates the same set of chains every time, or a different set of chains every time. The two simple rules to follow that guarantees the creation of the same set of chains for a given data dependence graph is:

1. Pick the first scheduled vertex that is legible to be the start of a chain every time a chain is created.
2. Pick the longest outgoing edge when there are multiple outgoing edges, where the longest is defined as the longest number of vertices between the two vertices in the schedule.

3.5 Chain Overlap

After identifying the chains of a data dependence graph, the next step is to verify if there is an overlap between these chains and where the overlaps occur. This is an important metric when trying to reduce the register pressure of a sequence of instructions, as some of the chains can share a register if they definitely do not overlap with one another. A scheduler can use this information to decide on how many registers a given data dependence graph needs to fully schedule the sequence of instructions and which chain can be assigned to which register.

Overlapping is defined as when two chains C_i and C_j have the property that no matter how the operations/vertices in the dependency graph are scheduled, the dependences in the graph always overlap. This can be summarized as the two chains C_i and C_j share at least one vertex with one another and that they also can be linked with one edge between any one vertex in C_i and any one vertex in C_j. An example of overlapping can be seen in Figure 3.5, where the dependency graph's vertices can be assigned to two chains, which are:

$$C = (V_1, V_2, V_3), (V_1, V_3)$$

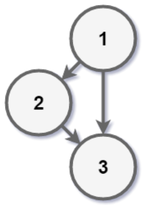


Figure 3.5: A Dependency Graph with Overlapping Chains

C_1 and C_2 definitely overlap with each other.

Non-Overlapping is defined as when two chains C_i and C_j have the property that no matter how the operations/vertices in the dependency graph are scheduled, the dependences in the graph never overlap. This is the opposite of overlapping as the two chains C_i and C_j must share no vertices with one another and that they also cannot be linked with one edge between any one vertex in C_i and any other vertex in C_j. An example of non-overlapping can be seen in Figure 3.6, where the dependency graph's non-overlapping chains are:

$$C = (V_1, V_3, V_5), (V_2, V_4, V_6)$$

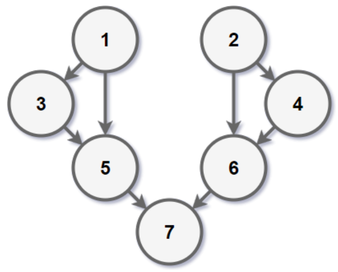


Figure 3.6: A Dependency Graph with Non-Overlapping Chains

These are the only two chains within the dependency graph that definitely do not overlap, these chains must be generated using the minimum chain algorithm. Note; using different rules when picking the chains in the minimum chain algorithm could result in different chains and therefore different overlapping/non-overlapping between these chains.

An algorithm is needed to calculate the overlap/non-overlap between each of the chains C_i and C_j, where $j > i$. The algorithm must compare each edge of a chain and check whether or not that edge's vertices exist in any other chain and whether there is a single edge that connects this vertex to a vertex within the other chain. If this condition holds true, the current chain and that chain are said to be definitely overlapping. Otherwise, both chains are said to be definitely non-overlapping.

The chain overlap graph is now coloured such that no two adjacent vertices share the same colour. The act of assigning a register to a chain can also be called colouring a chain.

3.5.1 Minimum Colours

If two chains are definitely non-overlapping they can share a colour, this is useful when considering the minimum number of colours a given data dependence graph requires. Since, not all chains can be scheduled, as their first vertices might have a dependence on another chain, it is non-trivial to calculate the minimum colours of a dependency graph before the scheduler has begun scheduling. However, the maximum number of colours needed is equal to the number of chains. An optimal solution will use the least possible number of colours. The scheduler knows that once it reaches the maximum number of colours, there is nothing for it to do as the schedule cannot be made more efficient. An estimate of the minimum colours in a non-trivial chain overlap graph can be initialized with $min = 3$ if there is no obvious solution.

The scheduler will need to start scheduling using the minimum colours. If it is impossible to create a valid schedule with the minimum colours, then the scheduler will need to try again with the minimum colours incremented. The scheduler will need to repeat this process until it either arrives at a valid schedule or it reaches the maximum number of colours. At this point the re-ordering of the original schedule is no longer necessary, as the original already has it's register pressure at the lowest that the data dependence allows.

3.6 Scheduling for Reduced Register Pressure

Perhaps the most important part of the entire process, scheduling for reduced register pressure, is also the biggest and most complicated step involved in the reducing register pressure process. To reduce the register pressure of a given set of instructions, a scheduler must be designed such that it re-orders the set of instructions to use as few registers as possible. When a processor needs more registers than it has access to, it causes a number of registers to spill. Reducing the register pressure improves the performance of the execution of this set of instructions, thereby reducing the number of registers spilled. Register spills cause a decrease in performance for a processor. These register spills are sometimes avoidable by the re-ordering and re-naming of registers in a set of instructions. This re-ordering preserves the functional output of a set of instructions such that the data dependences are adhered to and the overall flow of the instructions remains unchanged.

3.6.1 The Scheduler

The scheduler is a non-trivial heuristic solution. In this paper the following is a proposed solution that tries to efficiently schedule an instruction schedule aiming to reduce the register pressure.

The scheduler requires a target colour count to be set. This is the target number of registers that the scheduler will try to devise a solution around. It is possible that a solution for the target amount is not possible. The scheduler will then do it's best and preserve the original schedule as much as possible. The process can be re-run using a higher target number if desired.

The first thing the scheduler does is it finds all start vertices and picks the first scheduled one. A start vertex is a vertex that has no incoming edges. This vertex is allocated to the first available register. The scheduler then enters an iterative process that performs the following steps:

It finds the next suitable release vertex. A release vertex is a vertex that is at the end of one of the chains created for the dependency graph. A suitable release vertex has a complex definition but can be summarized by the following:

- The incoming vertices are able to be scheduled.
- The incoming vertices are able to be overwritten[*] by scheduling the release vertex.
- That at least one of the incoming vertices's outgoing vertices is released upon scheduling the release vertex.

[*]Overwritten in this context indicates that a vertex/register is no longer live i.e. it no longer has any vertices that haven't been scheduled that still depend on this vertex/register. This register can now be re-used by a new vertex.

Once the scheduler has picked a release vertex it begins the process of trying to traverse the data dependence graph using the chains to reach the release vertex. This is again an iterative process where at each iteration a new vertex is scheduled. The process can be summarized as follows:

1. The best chain must be found for the given release vertex.
2. The latest unscheduled vertex of the chain is fetched.
3. The scheduler either schedules the vertex in a register that can be overwritten upon scheduling the vertex, or in a new register if there are still available colours that fall within the target colour count.

The best chain is defined as the chain that contains the release vertex or is the chain currently required to traverse the dependency graph and advanced the scheduler by one iteration. This means the best chain for a given release vertex might change between other chains in an effort to reach the release vertex. An incoming vertex of one of the elements of the chain might not be scheduled and therefore the best chain is the chain that schedules that incoming vertex before continuing down the chain that contains the release vertex.

Once the scheduler has arrived at the release vertex, a new suitable release vertex must be chosen and the iterative process is continued until all the vertices have been scheduled. At which point the scheduler has completed it's task.

3.6.2 Deadlock

A deadlock is a situation in which the scheduler enters a state from which it can no longer make any progress towards completing the schedule with the allocated amount of registers. There are two types of deadlock: avoidable deadlock and unavoidable deadlock. A deadlock is an avoidable deadlock if the occurrence of deadlock depends on the order in which the operations are scheduled by the scheduler; otherwise the deadlock is unavoidable.

3.6.3 Avoiding Deadlock

To circumvent an avoidable deadlock, the scheduler needs to pick a suitable release vertex as described above. From all the possible release vertices there needs to be a guarantee of the release of register resources that happens such that deadlock does not occur. If by scheduling a release vertex that doesn't free at least one register or uses no more than the current register count, then that release vertex is not a suitable release vertex and another must be chosen. If no such release vertex exists then there is no way to avoid deadlock. The scheduler must try again with an increased target colour count. This process can be repeated until the scheduler manages to produce a schedule without deadlock. The expectation is that the new schedule uses less registers than original schedule.

3.7 Randomly Generated Data Dependence Graph

A randomly generated data dependence graph is a directed graph

$$G = (V, E)$$

$V \subseteq V^*$ where V* is a set of randomly generated vertices. The number of elements in the set V* follow a discrete uniform distribution between natural numbers V_{Min} and V_{Max} and is given by

$$V_n^* \sim U[V_{Min}, V_{Max}], V_{Max} > V_{Min}$$

E is a set of edges that randomly connects vertices from the set V* to one another. The process of randomly connecting these vertices is described in the following. Begin with Vertex V_i with $i = 1$, the second element of the V* set. The number of trials follows a discrete uniform distribution between 0 and E_{Max} and is defined as

$$E_n \sim U[0, E_{Max}], i > 0$$

E_{Max} is the maximum allowed connections for all of the vertices and unless otherwise specified, is a constant natural number. Now randomly connect V_i to a random preceding vertex V_j, where $j < i$, repeat this E_n times. If $E_n = 0$ then no connection is made. The probability of connecting any V_j to V_i is given by

$$P(E_{ij}) = \frac{1}{i}, i > 0$$

This means V_i has an equal probability of connecting to any one of the previous vertices.

After E_n connections are made for V_i, proceed to the next vertex V_{i+1} and repeat this process until $i = V_{Max} - V_{Min}$. Once this process is complete and all the edges are connected, any vertices with at least one connection are added to the set V as these vertices have some dependence, where as the set $V^* \setminus V$ contains all the vertices that do not have any dependence on any other vertex from the set V*.

This kind of randomly generated graph is similar to the Erdős–Rényi random graph model in graph theory [2]. Where some of the differences include having a limit E_{Max} on the number of connections between vertices and removing non-connected vertices from the graph.

Chapter 4

Examples for Reducing Register Pressure of Data Dependence Graphs

4.1 Worked Examples

In this chapter, a number of different data dependence graphs will be looked at, each with their own interesting properties that a comprehensive scheduler needs to take in account, when scheduling to reduce register pressure.

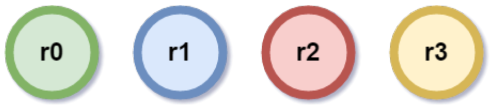


Figure 4.1: Register Colours for Worked Examples

The registers will follow a convention of being assigning a different colour for each required register. This can be seen in Figure 4.1. "Green" is assigned to "r0", "Blue" is assigned to "r1", etc... The important thing to note is that the use of different colours is to distinguish registers from one another, it is not necessary to remember which colours are associated to which register.

4.2 Mirror Formation

In this example, called "Mirror Formation", the current scheduling order requires four registers to schedule all instructions. However, if the scheduler rearranges some of the instructions, it can in fact be scheduled using only three registers. This presents a basic case for which the process of reducing register pressure can be simply shown and outlined. Using these worked examples as a guideline, the key decisions made while scheduling are discussed while the overall process remains simplified.

4.2.1 Instruction Schedule

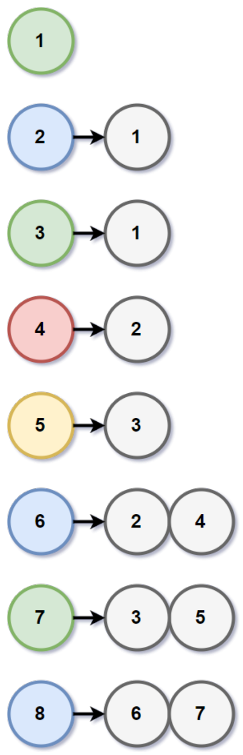


Figure 4.2: Mirror Formation: Instruction Schedule

The instruction schedule is shown in Figure 4.2. As previously mentioned, these vertex bubbles represent a sequence of assembly commands/instructions. The coloured vertex bubbles correspond to an allocated destination register.

Using the colour convention of registers seen in Figure 4.1, each particular instruction is coloured to show the original assigned register, before any rescheduling has occurred. In Figure 4.2 the 1st vertex (V_1) is coloured green and therefore scheduled initially with register r0. White or uncoloured vertex bubbles on the right are the operands of the instruction, and as such this means they are the dependencies of the vertices on the left. A coloured vertex bubble without any outward arrows, such as V_1, has no dependence on any other vertex. Coloured vertex bubbles with an outward arrow are the opposite and have a dependence on one or more other vertices. This is true for this example for every instruction except the first, where for example V_2 has one dependency and the V_6 has two dependencies.

In this example, it can be seen that with this current ordering of instructions, four registers are needed to schedule all instructions. It is worth remembering that a vertex cannot be scheduled before its dependencies. Once a vertex has been scheduled and no future vertices depend on this vertex, the register can be freed and used by another vertex. An example of this is V_3. The register V_3 uses the same as V_1, because no other vertex has any dependence on V_1, once V_3 is scheduled.

4.2.2 Data Dependence Graph

The dependence of the instruction schedule can be represented in a data dependence graph as in Figure 4.3. The graph is represented as

$$G = (V, E)$$

V is the set of vertices representing each instructions from the instruction schedule and E is the set of edges, representing the data dependence between the instructions. Each vertex is represented as a bubble with its scheduled number or vertex index inside. This number corresponds to the original sequencing order of the vertices. The edges are one directional arrows that represent a dependence between the vertices. Looking at V_1, it can be seen that V_2 and V_3 depend on V_1. Conversely, looking at V_8, it can be seen that it has a dependence on V_6 and V_7.

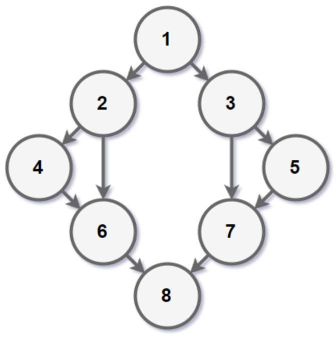


Figure 4.3: Mirror Formation: Data Dependence Graph

4.2.3 Chains

A chain is a sequence of adjacent edges that are extracted from the dependency graph. They can be represented as

$$C = (v_1, v_2), (v_2, v_3), \ldots, (v_{k-2}, v_{k-1}), (v_{k-1}, v_k)$$

Chains are formed such a way that every edge in the dependency graph is contained within exactly one chain. Using the minimum chain algorithm, chains are formed and highlighted within the dependency graph, this can be seen in Figure 4.4. Each highlighted colour is different in such a way that every chain has a different colour. These colours are not related to the register colours in Figure 4.1. From this diagram, it can be seen that the minimum amount of chains needed is four. The chains are separated out in a graphical form in Figure 4.5.

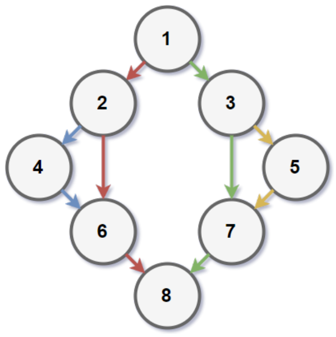


Figure 4.4: Mirror Formation: Data Dependence Graph Highlighted

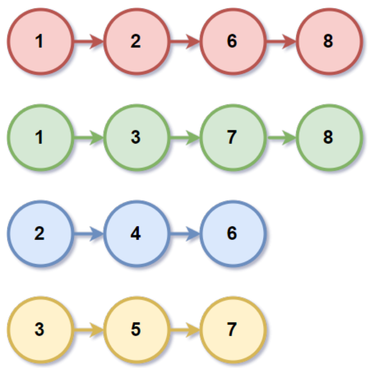


Figure 4.5: Mirror Formation: Chains for Data Dependence Graph

4.2.4 Chain Overlap

The overlap graph, shown in Figure 4.6, is calculated from Figure 4.5, where there is an overlap of two chains, there is an arrow connecting the two chains. It can be seen in this example that chain C_1 and C_2 overlap with each other.

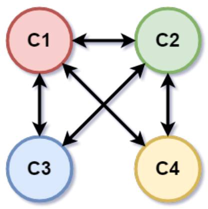


Figure 4.6: Mirror Formation: Overlap of Chains

Chains C_3 and C_4 are the only pair of chains that do not overlap with each other, see Figure 4.7. Each chain by itself would require a single register. Since this example contains four chains, it would normally indicate that in order to schedule all of the chains, the scheduler would require four registers. However, as there are chains that do not overlap, they can share a register between themselves. They can also be ordered in such a way that one of the chains can be scheduled to execute before the other one executes.

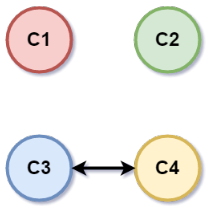


Figure 4.7: Mirror Formation: Non-Overlap of Chains

4.2.5 Start and Release Vertices

A start vertex is a vertex that starts at position 0 in a chain i.e. the first vertex in a chain. This is useful to note because a start vertex indicates that a register resource must be allocated and retained once the start vertex is scheduled, and remain so until all the vertices in a chain have been fully released. A release vertex is the last vertex in a chain i.e. the vertex at which the chain is fully released and all registers allocated to it can be freed, as long as they do not have any other vertices depending on them. The start vertices can be seen in Figure 4.8 and the release vertices can be seen in Figure 4.9.

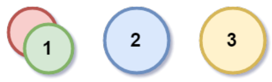


Figure 4.8: Mirror Formation: Start Vertices of Chains

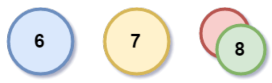


Figure 4.9: Mirror Formation: Release Vertices of Chains

4.2.6 Scheduling

After collecting all the previous information from the instruction schedule, the scheduler can begin scheduling a suitable solution to the problem. It may be the case that the scheduler cannot arrive at a solution that reduces the register pressure, but in this example, this is not the case.
In order to arrive at a suitable solution, the scheduler is required to make a

number of important decisions.

The scheduler is first required to prioritize one of the release vertices. This assists in determining which chain and vertex to pick for each schedule slot as there are numerous criterion that need to be considered when scheduling a given instruction. There are three choices to start with as seen in Figure 4.9. Choosing either V_6 or V_7 does not matter in this example as the dependency graph is symmetrical and scheduling either to the left or to the right produces a mirrored result of the other. This means between the two choices, the scheduler will often pick the one that was originally first scheduled, which in this case, is V_6. The other option is for the scheduler to prioritize is V_8. This isn't a viable option even though two of the four chains have a release vertex at V_8, as both release vertices V_6 and V_7 need to be scheduled before V_8. This means V_8 cannot be selected as the priority release vertex for this step. V_6 is therefore selected as the priority release vertex target.

Once a priority release vertex is identified, the scheduler must create the best course to navigate the dependency graph to the release vertex target. It therefore picks a chain that follows the logic: a chain that contains the release V_6 in it and where that chain is immediately able to be scheduled. This means C_1 is chosen and V_1 is scheduled as it is the first vertex in the chain (1 register used). V_2 can next be scheduled as it is the next vertex in C_1 (2 registers used). The next vertex that the scheduler wants to schedule is V_6, however, V_6 cannot be immediately scheduled as it is missing one of its required vertices V_4. The scheduler then schedules V_4 (3 registers used). V_6 can now be scheduled. This releases a register that was used by C_3, reducing the number of registers in use (2 Registers Used). Now that the scheduler is at the priority release vertex target, a new one is required to be selected.

V_7 is selected as the priority release vertex target because it is the only remaining one that can be selected. V_8 still cannot be selected for the same reasons as previous stated. There are two live registers currently operative, which hold V_1 and V_6. Following the procedure for scheduling V_2 and beyond, the process is identical because of the symmetry of the dependency graph. This means V_3 is Scheduled (2 Registers Used), V_5 is scheduled (3 Registers Used) and V_7 is scheduled (2 Registers Used). The new priority release vertex target is now V_8 as there are no other options. Both of it's dependencies have been scheduled, therefore V_8 is scheduled (1 Register Used). All of the vertices have now been scheduled and the scheduler has completed it's task. The final output/reduced register pressure solution can be seen as in Figure 4.10.

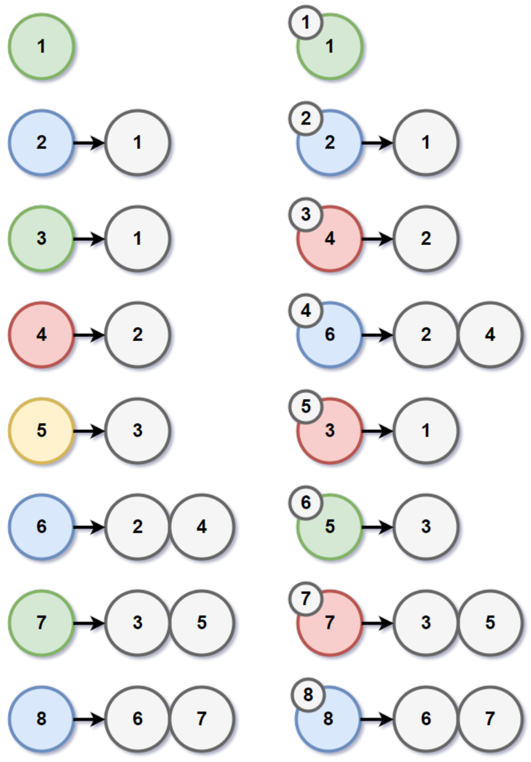


Figure 4.10: Mirror Formation: A Comparison of the Input versus Output

In Figure 4.10 it can be clearly seen that the scheduler has managed to re-order and re-colour the instruction schedule in such a way that it only uses three registers, instead of the original four registers. V_4 and V_6 have moved before V_3 and while both keep the colour they were originally assigned: V_3, V_5 and V_7 now use different colours. This example shows that it is possible to re-order and re-colour an instruction schedule to reduced the register pressure.

4.3 Diamond Formation

In this example, called ”Diamond Formation”, the instruction schedule is omitted from having its own section, as all the information it presents is now present in the data dependence graph, however, the schedule is present in the final schedule output in Figure 4.15. The unaltered instruction schedule requires four registers, but if the scheduler rearranges some of the instructions this can be reduced to only three registers. This example is more interesting than the previous ”Mirror Formation”, as some interesting decisions arise when scheduling, the dependency graph contains more vertices and edges, and picking a priority release vertex is not as trivial as in the ”Mirror Formation”.

4.3.1 Data Dependency Graph

The data dependency graph is constructed in Figure 4.11. In addition the dependency graph's edges are highlighted to represent the chains constructed and there are seven different chains (each coloured). There is a small bubble located at the start of each of the chains in the top left side of the vertex bubbles. The content is in the form C_i where i is the chain index.

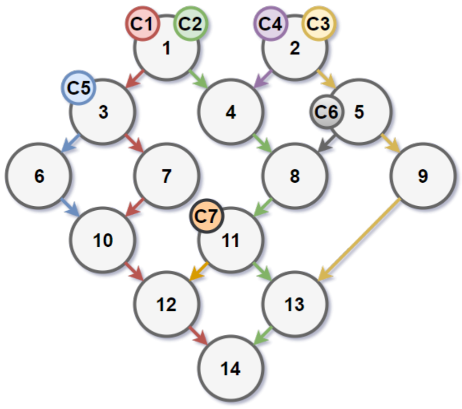


Figure 4.11: Diamond Formation: Data Dependence Graph Highlighted with Chain Indexes

4.3.2 Chain Overlap, Start and Release Vertices

The overlap is calculated and shown in Figure 4.12. While trying to colour the chains, the number of colours used will be calculated as four. This is not true however. It can be seen in this example that no chain except C_1 overlaps with C_5. However in this example, the scheduler can share registers between C_1 and C_5 as C_1 must be scheduled before C_5 can be scheduled and since C_5 concludes before C_1, the registers used by C_5 remain live, but once C_2 has been partially scheduled, there is only one live register used by both chains C_1 and C_5. Since C_5 does not overlap with any other chain, the freed register used by completing the dependence on V_1 can be shared with any of the other chains. Now that C_5 has been scheduled, the remaining vertices can be scheduled in any legal order, as they all overlap and there is no more register pressure reduction that can take place. A total of three registers is needed to finish traversing the dependency graph and chains. The start and release vertices are shown in Figure 4.13 and Figure 4.14 respectively.

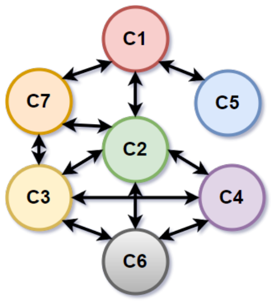


Figure 4.12: Diamond Formation: Overlap of Chains

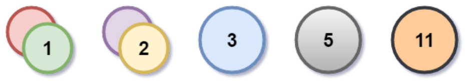


Figure 4.13: Diamond Formation: Start Vertices of Chains

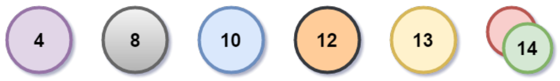


Figure 4.14: Diamond Formation: Release Vertices of Chains

4.3.3 Scheduling

As with the previous example, a priority release vertex is needed to be selected from the available release vertices. In this example, it is really important for the first release vertex to be selected in the correct order as there is only one vertex that yields a correct solution for using three registers to complete. This is release V_{10}. The other two release vertices that may be considered by the scheduler would be V_4 and V_8 as they appear first sequentially in the release vertex list. Selecting V_4 is incorrect, so whilst it may appear correct while scheduling the first few following vertices, it will encounter problems when it has to schedule past scheduling V_4. The reason as to why it cannot be chosen as the priority release vertex is that both V_1 and V_2 are required for scheduling V_4, but neither can release their register and cannot be freed as C_1 and C_3 still require them to be live. Selecting V_8 follows the same logic as it would require V_4 to be scheduled.

The scheduler selects V_10 as it's first priority release vertex. It then proceeds to schedule the vertices: V_1 (1 register used), V_3 (2 registers used), V_7 (3 registers used), V_6 (3 registers used) and then V_{10} (2 registers used). The next priority release vertex is V_4. Since, scheduling V_4 now releases the register that V_1 had been using up until this point.

The scheduler selects V_4 as it's second priority release vertex. It then proceeds to schedule the vertices: V_2 (3 registers used) and V_4 (3 registers used). The scheduler then selects V_8 as it's third priority release vertex and proceeds to schedule the vertices: V_5 (3 registers used) and V_8 (3 registers used).

The three live values currently are: V_{10}, V_8 and V_5. The scheduler selects V_{12} as it's fourth priority release vertex. It then proceeds to schedule the vertices: V_{11} (3 registers used) and V_{12} (3 registers used). The three live values currently are: V_{12}, V_{11} and V_5. The scheduler selects V_{13} as it's fifth priority release vertex. It then proceeds to schedule the vertices: V_9 (3 registers used) and V_{13} (2 registers used).

Lastly, the scheduler picks and schedules V_{14} as it's sixth priority release vertex (1 register used). All of the vertices are now scheduled. The final schedule can be see as in Figure 4.15.

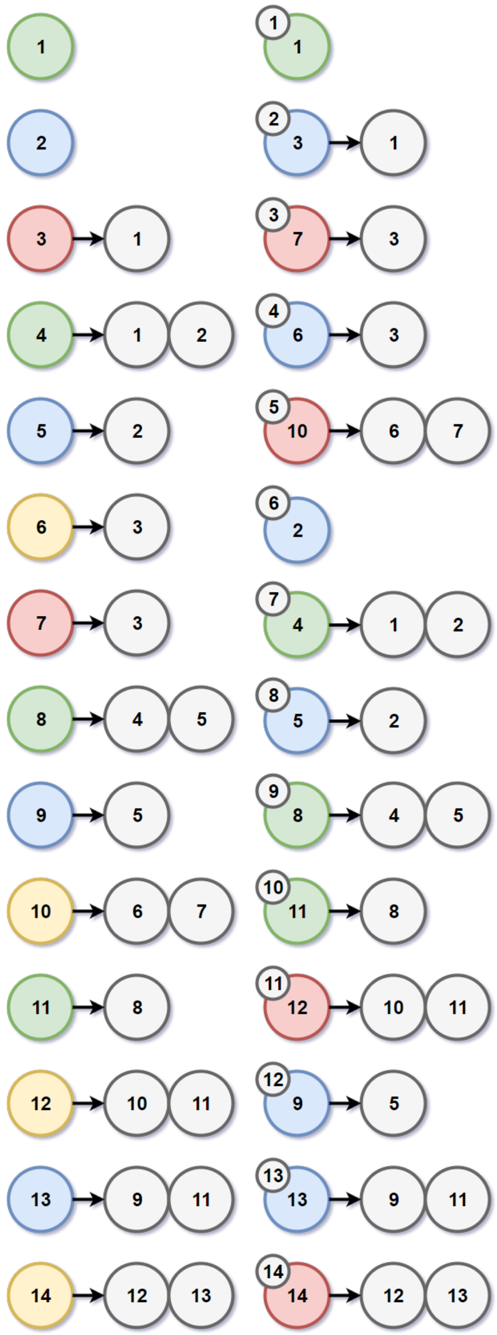


Figure 4.15: Diamond Formation: A Comparison of the Input versus Output

4.4 Rhombus Formation

In this example called ”Rhombus Formation”, only the comparison of the instruction schedule and the re-ordered schedule is shown as it is quite often the case that after scheduling, the scheduler doesn’t manage to reduce the register pressure but the output will have most likely been altered. As the scheduler will often travel down large chains, groups of dependences within the center of the chains will likely be nearer to one another.
In this case some decision needs to be made about whether to keep the new schedule or the old one, as they both have the same register pressure.

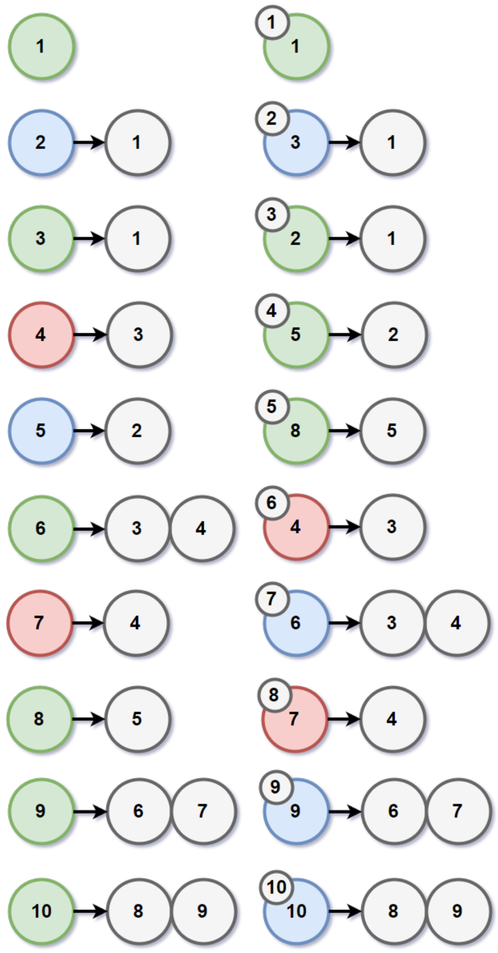


Figure 4.16: Rhombus Formation: A Comparison of the Input versus Output

Chapter 5

Research Findings

5.1 Random Data Dependence Graph

Random data dependence graphs were generated and run through the scheduler thousands of times to find the average reduced register pressure gained by using the scheduler. There are a number of variables used that have an effect on the reduced register pressure gained, some of these variables are able to be controlled and are listed in Table 5.1.

Term	Description
V*	The set of randomly generated vertices
V_{Max}	The maximum allowed number of vertices in the set V*
E_{Max}	The maximum allowed connections between vertices of the set V*
N	The number of trials used in calculating the averages
$V_{Min} = 5$	The minimum allowed number of vertices in the set V*
Initial	The initial amount of registers used, before the scheduler
Final	The final amount of registers used, after the scheduler
Gain	The gain from using the scheduler, Gain = Initial - Final

Table 5.1: The Variables used in generating the results

5.2 The Effect of Increasing the Number of Vertices

V_{Max}	Vertices	Edges	Chains	Final	Initial	Gain
30	15.964	17.128	11.14	8.374	8.837	0.463
60	27.805	31.178	19.497	14.9	15.191	0.291
120	53.089	61.426	37.287	28.527	28.685	0.158
180	79.777	93.413	55.943	42.843	42.928	0.085
270	113.191	133.929	79.575	60.587	60.671	0.084
360	151.049	178.924	105.743	80.787	80.822	0.035
720	306.036	365.033	214.772	163.627	163.663	0.036
1440	599.469	716.953	420.117	320.38	320.399	0.019

Table 5.2: The Effect of Varying V_{Max}, N = 1000, $E_{Max} = 3$

5.3 The Effect of Increasing the Number of Trials

N	V_{Max}	Vertices	Edges	Chains	Final	Initial	Gain
1000	30	15.964	17.128	11.14	8.374	8.837	0.463
10000	30	15.7912	16.7992	11.0391	8.3043	8.7935	0.4892
1000	60	27.805	31.178	19.497	14.9	15.191	0.291
10000	60	28.4313	31.8391	19.8873	15.2803	15.5323	0.252
1000	120	53.089	61.426	37.287	28.527	28.685	0.158
10000	120	53.8229	62.2409	37.7395	28.947	29.078	0.131
1000	180	79.777	93.413	55.943	42.843	42.928	0.085
10000	180	78.8753	92.3023	55.3206	42.383	42.3711	0.0839

Table 5.3: The Effect of Varying N for a given V_{Max}, $E_{Max} = 3$

5.4 The Effect of Changing the Maximum Dependencies

E_{Max}	Vertices	Edges	Chains	Final	Initial	Gain
2	12.9731	9.306	6.4697	3.3086	6.4697	3.1611
3	15.7912	16.7992	11.0391	8.3043	8.7935	0.4892
4	17.2916	24.7255	15.5018	9.9835	10.1332	0.1497

Table 5.4: The Effect of Varying E_{Max}, $V_{Max} = 30$, N = 10000

Chapter 6

Discussion of Results

It is clear from the results that running the scheduler for randomly generated data dependency graphs will, on average, net a gain of reduced register pressure of some amount. This means that there is never a case where the scheduler performs worse in regard to reducing the register pressure on an instruction schedule, as it can only have the same register pressure or improve the result. This means that scheduling for reduced register pressure is always beneficial.

It is also possible that the data dependency graphs that are randomly generated do not accurately or entirely represent "real world" data dependency graphs and as a result of this there may be further increases or decreases in the gain of reduced register pressure by scheduling. In addition to this, there may be several improvements to the design of register pressure reduction schedulers that are not fully explored in this paper.

6.1 Reduced Pressure and the Number of Vertices

The results seen in Table 5.2 show the effect that increasing the number of vertices has on the reduced register pressure gained by running the scheduler. The number of trials is set to 1000 throughout each of the scheduling runs, whilst the maximum number of allowed connections between vertices is set to 3.

The effect of increasing the number of vertices in the randomly generated graphs has an overall decrease in the gain of reduced register pressure. This is somewhat intuitive, as with the increase of the number of vertices, there is also an increase in the number of edges between the other vertices. This in turn increases the number of chains required to fully traverse each edge within the data dependency graph. As the number of chains increases on average, so does the number of registers needed. This means as the data dependency graph becomes more complex with the increase in vertices. The scheduler also has a more difficult time reducing the register pressure as the vertices later in the instruction schedule still very much depend on vertices earlier on in the schedule.

While the effect of running the scheduler on the data dependency graphs has diminishing returns on graphs with a large amount of vertices, there is always some gain to running the scheduler as it does on average produce a net gain of reduced register pressure. One could argue that for large graphs, the scheduling process could be skipped as the gain yields are so low or negligable.

6.2 The Number of Trials within the Experiments

The results seen in Table 5.3 show the effect that the number of trials have on the averages collected during the experiment. The number of trials is varied between 1000 and 10000 throughout each of the scheduling runs. The maximum number of allowed connections between vertices is set to 3 throughout. The maximum number of vertices generated is also varied for a given number of trials.

The effect that the number of trials which have exceeded 1000 is small, as the difference between the Initial and the Final is only a few decimal points. This is useful because the time taken to simulate a large amount of trials drastically increase as the number of trials increases. As the difference is small between two sets of results for each number of trials, we can assume that for the number of trials set to 1000 is a decent reference point for the generated averages that have been achieved.

6.3 Reduced Pressure and the Maximum Dependencies

The results seen in Table 5.4 show the effect that increasing the maximum allowed connections or dependencies between vertices has on the reduced register pressure gained by running the scheduler. The number of trials is set to 10000 in this case. The maximum number of vertices is set to 30 throughout.

It can seen that the effect of increasing the number of allowed connections has a similar decrease in the gain of reduced register pressure, as with the number of vertices. With a larger amount of connections, there will be a larger amount of edges in the data dependence graph. This in turn increases the number of chains and therefore the number of needed registers. While on small graphs, there is always cause for scheduling for reduced register pressure, a case can be made that it is always worthwhile, no matter the limit on the maximum number of connections, as realistically this number is small.

A vertex in the context of an instruction has a limited amount of data dependence. This is why there is a need to put a limit on the maximum number of dependencies or vertex connections. Typical instructions vary in the number of dependencies from 0-2. It is worth noting that instructions in an instruction set do not appear with the same frequency as one another, as instructions like "MOV" are among the most frequently used instructions and it only requires one data dependency. This could influence future random data dependency graphs that are tested with the scheduler. See [7] for some statistics on some of the most frequent types on instructions used, that should commonly appear in data dependency graphs. In this experiment the number of connections a vertex has is uniformly distributed and this may not reflect "real world" data dependency graphs.

6.4 Register Spills

Register spills in the context of these results shows some opportunity for improvement. As there may not be always be a net gain upwards of one register, there will definitely be cases in which reducing the register pressure will have a positive effect on reducing the register spills. Shown in Table 5.4, the connections between vertices have the largest impact on the register gain going from $E_{Max} = 2$ to $E_{Max} = 3$. As previously mentioned, the random data dependency graphs may lack some ”real world” qualities but it is likely that a ”real world” dependency graph will lie somewhere in between $E_{Max} = 2$ and $E_{Max} = 3$. As the more commonly used instruction [7] have fewer dependencies than the less frequently used instructions. This would impact the average number of connections a randomly vertex would need to follow in generating a more ”real world” randomly generated data dependency graph.

Chapter 7

Conclusion

7.1 Final Comments

This paper has shown that there is benefit in the compiler trying to re-order instructions to reduce the number of simultaneously live values, otherwise known as reducing the register pressure. As a block of instructions or instruction schedule have the maximum register pressure to begin with, re-ordering the instruction with the aim of reducing the register pressure will either show no improvement in the final result (in which case there is no harm having validated this) or some improvement with the register pressure. This means that the number of simultaneously live values has decreased and as such there will be performance improvements if the reduction in the simultaneously live values is enough to overcome register spillage. Other work done on reducing register pressure can be seen [3, 4].

7.2 Future Work

Future work could entail a lot more investigation on the randomly generated data dependency graphs, as there is still a lot that could be argued over their "real world" nature. For example, taking the distribution of the frequency of instructions used [7], could model the probability of connecting vertices when generating the dependency graphs. Otherwise, other models could be used, such as, vertices tend to be connected to other vertices closer in the schedule than those further away (and vice versa) and such the probability of connecting a vertex to one that is far away should be different to a vertex that is closer in the schedule.

More complex instruction schedulers could also be designed to handle more specific cases, where instead of abstracting away from the actual instruction into the instruction name and data dependencies, their opcodes could be taken into account as they may be various delays for each opcode (as memory accessing instructions are slower than regular arithmetic instructions for example) or situations where specific instructions must be handled in specific ways as to optimize a schedule.

Bibliography

[1] ARM. *ARM Compiler toolchain Assembler Reference*. URL: http://infocenter.arm.com/help/topic/com.arm.doc.dui0489f/DUI0489F_arm_assembler_reference.pdf.

[2] P Erdos and A Renyi. "On random graphs I." In: *Publ. Math. Debrecen* 6 (1959), pp. 290–297.

[3] R. Govindarajan, Chihong Zhang, and Guang R. Gao. "Minimum Register Instruction Scheduling: A New Approach for Dynamic Instruction Issue Processors". In: *Languages and Compilers for Parallel Computing: 12th International Workshop, LCPC'99 La Jolla, CA, USA, August 4–6, 1999 Proceedings*. Ed. by Larry Carter and Jeanne Ferrante. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 70–84. ISBN: 978-3-540-44905-8. DOI: 10.1007/3-540-44905-1_5. URL: http://dx.doi.org/10.1007/3-540-44905-1_5.

[4] R. Govindarajan et al. "Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures". In: *IEEE Transactions on Computers* 52.1 (Jan. 2003), pp. 4–20. ISSN: 0018-9340. DOI: 10.1109/TC.2003.1159750.

[5] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2011. ISBN: 9780123838735. URL: https://books.google.ie/books?id=gQ-fSqbLfFoC.

[6] Iowa-State-University. *Data Hazards*. URL: https://web.cs.iastate.edu/%7B~%7Dprabhu/Tutorial/PIPELINE/dataHaz.html.

[7] Peter Kankowski. *Which instructions and addressing modes are used most often*. URL: https://www.strchr.com/x86_machine_code_statistics.

[8] Charles Kozierok. *Processor Instructions*. 2001. URL: http://www.pcguide.com/ref/cpu/arch/int/inst-c.html.

[9] LLVM. *LLVM Language Reference Manual*. URL: http://llvm.org/docs/LangRef.html.

[10] Sparsh Mittal. "A survey of techniques for designing and managing CPU register file". In: *Concurrency and Computation: Practice and Experience* (2016).

[11] G. O'Regan. *A Brief History of Computing*. SpringerLink : Bücher. Springer London, 2012. ISBN: 9781447123590. URL: https://books.google.ie/books?id=QqrItgm351EC.

[12] D. Page. *A Practical Introduction to Computer Architecture*. Texts in Computer Science. Springer London, 2009. ISBN: 9781848822566. URL: https://books.google.ie/books?id=XH4sIpY1D70C.

[13] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2004. ISBN: 9780080502571. URL: https://books.google.ie/books?id=1lD9LZRcIZ8C.

[14] Webster Dictionary. *Register Spilling*. URL: http://www.webster-dictionary.org/definition/register%20spilling.

[15] R.J. Wilson. *Introduction to Graph Theory*. Longman Scientific & technical. Longman, 1985. URL: https://books.google.ie/books?id=1-nuAAAAMAAJ.

Appendix A

Appendices

Developed with Visual Studio 2015 in Visual C++.
Code used can be found at:
https://github.com/Coggroach/RegisterPressure
LLVM binary and source files are not included.
LLVM code can be found at:
http://releases.llvm.org/