Register Allocation and Spilling Using the Expected Distance Heuristic

by

Ivan Neil Burroughs
B.Sc., University of Victoria, 2006
M.Sc., University of Victoria, 2008

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

© Ivan Neil Burroughs, 2016
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Supervisory Committee

Professor R. Nigel Horspool, Supervisor

(Department of Computer Science)

Professor Yvonne Coady, Departmental Member

(Department of Computer Science)

Professor Amirali Baniasadi, Outside Member

(Department of Electrical and Computer Engineering)

Dr. Paul Lalonde, Additional Member

(Google, Inc.)

Supervisory Committee

Professor R. Nigel Horspool, Supervisor
(Department of Computer Science),
Professor Yvonne Coady, Departmental Member
(Department of Computer Science)
Professor Amirali Baniasadi, Outside Member
(Department of Electrical and Computer Engineering)
Dr. Paul Lalonde, Additional Member
(Google, Inc.)

ABSTRACT

The primary goal of the register allocation phase in a compiler is to minimize register spills to memory. Spills, in the form of store and load instructions, affect execution time as the processor must wait for the slower memory system to respond. Deciding which registers to spill can benefit from execution frequency information yet when this information is available it is not fully utilized by modern register allocators.

We present a register allocator that fully exploits profiling information to minimize the runtime costs of spill instructions. We use the *Furthest Next Use* heuristic, informed by branch probability information to decide which virtual register to spill when required. We extend this heuristic, which under the right conditions can lead to the minimum number of spills, to the control flow graph by computing Expected Distance to next use.

The furthest next use heuristic, when applied to the control flow graph, only partially determines the best placement of spill instructions. We present an algorithm for optimizing spill instruction placement in the graph that uses block frequency information to minimize execution costs. Our algorithm quickly finds the best placements for spill instructions using a novel method for solving placement problems.

We evaluate our allocator using both static and dynamic profiling information for the SPEC CINT2000 benchmark and compare it to the LLVM allocator. Targeting the ARMv7 architecture, we find average reductions in numbers of store and load instructions of 36% and 50%, respectively, using static profiling and 52% and 52% using dynamic profiling. We have also seen an overall improvement in benchmark speed.

Contents

Supervisory Committee ii

Abstract iii

Table of Contents iv

List of Tables vii

List of Figures viii

1 Introduction 1
- 1.1 Thesis Outline . . . 4

2 Background 5
- 2.1 Intermediate Code . . . 6
  - 2.1.1 Static Single Assignment Form . . . 10
- 2.2 Register Allocation . . . 12
  - 2.2.1 Spilling . . . 13
  - 2.2.2 Spill Placement . . . 15
  - 2.2.3 Move Coalescing . . . 15
  - 2.2.4 Register Assignment . . . 16
  - 2.2.5 Calling Conventions . . . 16
  - 2.2.6 Architectural Issues . . . 17
- 2.3 Register Allocation Methods . . . 20
  - 2.3.1 Local Register Allocation . . . 20
  - 2.3.2 Graph Coloring . . . 21
  - 2.3.3 Linear Scan . . . 24
  - 2.3.4 Other Methods . . . 25

2.4 Complexity Of Register Allocation 26
2.5 Spill Code Placement 28
2.6 Move Instruction Coalescing 32
2.7 Beyond Register Allocation 35
2.8 Spill slots and the data cache 35

3 Register Allocation Using Expected Distance 38
3.1 Bélády's MIN Algorithm 39
3.2 Furthest Distance To Next Use 42
3.3 Distance Analysis 43
3.3.1 Liveness Probability 43
3.3.2 Expected Distance 45
3.3.3 Special Cases 46
3.4 Allocation 50
3.4.1 Input Register Allocation State 51
3.4.2 Allocation Within Basic Blocks 54
3.4.3 Spill Preferencing 58
3.5 Spill Placement 59
3.5.1 Problem Description 60
3.5.2 Solving 64
3.6 Move Coalescing 75
3.6.1 Algorithm 78

4 Evaluation 82
4.1 Spill Counts 83
4.1.1 Static Spill Counts 83
4.1.2 Dynamic Spill Counts 83
4.2 Implementation 84
4.2.1 ARMv7 General Purpose (Integer) Registers 86
4.2.2 ARMv7 Floating Point Registers 86
4.2.3 Calling Convention 87
4.2.4 32-bit Move Instructions 87
4.3 Benchmark 87
4.4 Results 88
4.4.1 Dynamic Profiling 90

4.4.2 Static Profiling 94
4.4.3 Nearest Next Use 97
4.4.4 Dead Edge Distances 102
4.4.5 Restricted Registers Experiment 107
4.4.6 Timings 111
4.4.7 Summary 116

5 Conclusion **119**
5.1 Future Work 121

Appendix **122**

A Iteration for Data Flow Analysis **122**

Bibliography **125**

List of Tables

4.1 ARMv7 General Purpose Integer Registers. Caller-save registers are in grey. SP and PC are not available for use. 86
4.2 ARMv7 Floating Point Registers. Caller-save registers are in grey. . . 86
4.3 Restricted set of ARMv7 General Purpose Integer Registers. Those marked in grey are not available to any compiler stage including the register allocator. 110

List of Figures

1.1 Simplified Compiler Diagram. 2

2.1 Splitting a live range with two definitions on a critical edge using a move instruction. Physical registers have already been assigned. . . . 7

2.2 Intermediate representation for the function written in C, and its corresponding control flow graph 8

2.3 SSA intermediate representation for the C function of Figure 2.2, and its corresponding control flow graph showing only the renumbered virtual registers. 11

2.4 Unnecessary moves due to merging control flow paths and the function calling convention where R0 and R1 are argument registers and R5 is preserved across the function call. 16

2.5 A register assignment that limits pipelining of instructions and another that does not. 18

2.6 Register Aliases on the a) Intel x86, and b) ARM VFP architectures. 18

2.7 Aliased register allocation showing the potential for interferences between registers of different sizes to cause spills. a) A poor allocation causing a spill, b) A successful allocation. "define f0" places the virtual register f0 into a physical register. "kill f1" frees the physical register of virtual register f1. 19

2.8 The stages of the Iterated Register Coalescing Graph Coloring allocator[38] as shown in [5]. 23

2.9 A spilled virtual register live range. 30

2.10 Example stack frame layout for the ARM architecture. The first four arguments are held in R0-R3. The function return address is held in the LR (R14) register. The SP (R13) register holds the stack pointer which points to spill slot 0. 35

3.1 The Nearest Next Use heuristic would choose to spill V6 at the Spill Point. Block lengths are shown underneath each block and distance to virtual register uses within the block are shown on top. Computed distances to virtual register uses are shown on the bar at the bottom. 41
3.2 Data Flow Equations for computing Liveness Probability. 44
3.3 Data flow equations for computing Expected Distance of virtual register v at both the top and bottom of basic block B. 45
3.4 Example control flow graph showing a variable that is live from $B2 \rightarrow B1$ but is dead on $B2 \rightarrow B3$. D corresponds to a definition and U to a use of R8. 46
3.5 Three examples having a an edge with a zero execution count. 49
3.6 Computing the input state for basic block $BB5$ with two registers available. The output states of blocks BB3 and BB4 show the locations of virtual registers in both physical registers and memory. These are used to compute the input state to BB5. Spills are inserted on edge $BB4 \rightarrow BB5$ in order to match locations at the input to $BB5$. 53
3.7 Allocation proceeds by modifying the register state and inserting spill instructions as necessary. Store and load instructions have been inserted into the block as required. The symbol `<k>` indicates a kill or last use of a virtual register. 54
3.8 An example control flow graph after allocation and before spill placement optimization. 61
3.9 The Store Graph for the control flow graph of Figure 3.8. 62
3.10 Store problem graph showing the minimum cut (dotted line) solution. 63
3.11 Simple edge reductions showing the flow graph, and its reduction: a) leaf nodes, b) sequential edges, c) back edges. 67
3.12 Showing an allocation of physical registers, including move instructions, for the input code with virtual registers. 76
3.13 Interference graph derived from the register assignment from Figure 3.12. Move related edges are shown as solid lines while interference edges are dotted. The cross-hatch live ranges are fixed physical registers. 77
3.14 Optimized interference graph showing the results of coalescing. Only one required move instruction is left. 77
3.15 Procedure for optimizing move related edges. 80

4.1 Percentage improvement in dynamically executed instructions of the DRA allocator compared to the LLVM Greedy allocator. Using dynamic profiling and distance multipliers. 91
4.2 Percentage improvement in dynamically executed instructions of the DRA allocator compared to the LLVM Greedy allocator. Using dynamic profiling and no distance multipliers. 95
4.3 Percentage improvement in dynamically executed instructions of the DRA allocator compared to the LLVM Greedy allocator. Using static profiling estimates and distance multipliers. 96
4.4 Percentage improvement in dynamically executed instructions of the DRA allocator when using Expected Distance over Nearest Next use. Both heuristics use dynamic profiling information and distance multipliers. 98
4.5 Percentage improvement in dynamically executed instructions of the DRA allocator when using Expected Distance over Nearest Next use. Both heuristics use static profiling estimates and distance multipliers. 101
4.6 Percentage improvement of the DRA allocator when the Dead Edge Distance Estimator is used to compute Expected Distance. The DRA allocator is using dynamic profiling and distance multipliers. 103
4.7 Percentage improvement of the DRA allocator, using the Dead Edge Estimator to compute Expected Distance, over LLVM. The DRA allocator is using dynamic profiling, distance multipliers. 104
4.8 Percentage improvement of the DRA allocator when the Dead Edge Distance Estimator is used to compute Expected Distance. The DRA allocator is using static profiling and distance multipliers. 105
4.9 Percentage improvement in dynamically executed instructions of the DRA allocator compared to the LLVM Greedy allocator. Both allocators are using a restricted set of registers. The DRA allocator is using dynamic profiling and distance multipliers. 108
4.10 Percentage improvement in dynamically executed instructions of the DRA allocator compared to the LLVM Greedy allocator. Both allocators are using a restricted set of registers. The DRA allocator is using static profiling and distance multipliers. 109

4.11 Percentage improvement in program running time when compiled using the DRA allocator over code compiled using the LLVM allocator. The DRA allocator is tested with both static and dynamic profiling. . . . 112
4.12 ARMv7 instruction sequence that a) stalls the processor, and b) its correction. 114

Chapter 1

Introduction

Programming language compilers are one of the most important tools for computer systems. They provide an interface between software developers and computer hardware by translating human readable source code into machine executable binary code. Their usefulness cannot be understated. They allow the programmer to concentrate on the problem they are trying to solve while leaving the management of the target hardware architecture details to the compiler.

The ability of the compiler to produce highly efficient and fast code that fully utilizes the target's architectural features is very important. It is well known in the hardware community that the success of a new processor is strongly tied to the availability of a good optimizing compiler for it.

Traditional compilers are normally constructed as a succession of transformations and are generally divided into three phases. The compiler front-end recognizes the plain text source code, checks that the language rules have been followed, and normally transforms it into machine independent intermediate code. The optimization stage performs analysis and optimization on the intermediate code in an effort to achieve the requested objectives whether program efficiency or size. Finally, the compiler back-end translates the machine independent code into machine dependent code for the target architecture. The back-end needs to understand the details of the target architecture as it selects native instructions and register arguments. Just-In-Time compilers incorporate the compiler back-end as a run-time stage for recompiling intermediate code during program execution.

One of the more important and, indeed, necessary tasks in the compiler back-end is the register allocation phase. The register allocator maps a potentially large number of program variables, and intermediate results, to a smaller number of physical

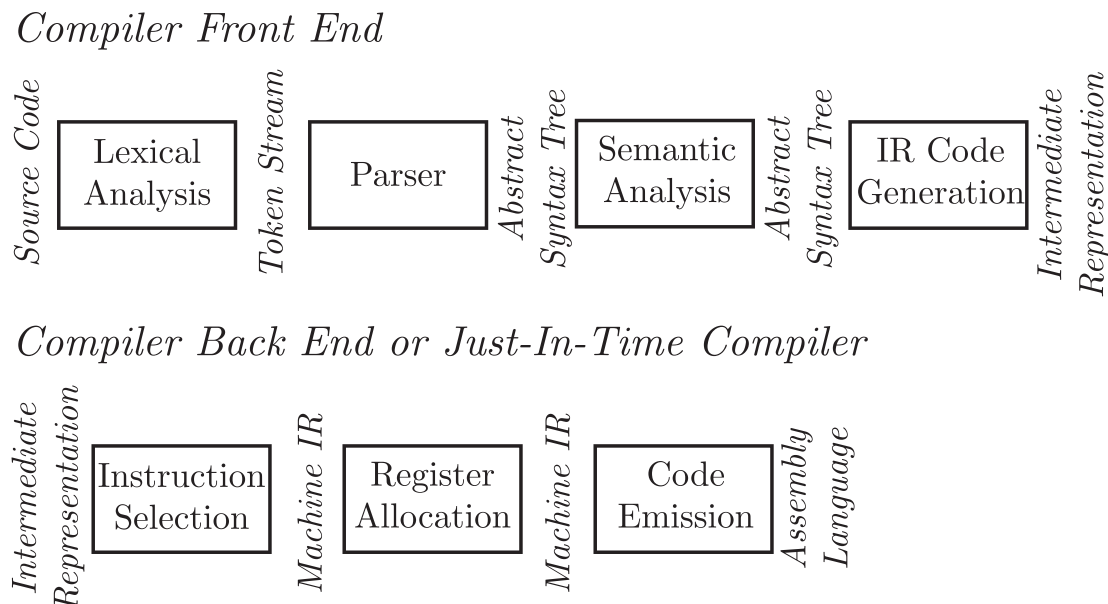


Figure 1.1: Simplified Compiler Diagram.

registers contained in the target processor. Code efficiency is tied to having instruction arguments available in a processor register when they are needed. However, it is not always possible to fit all the variables that are live at a given point in the program into a limited number of registers. Some variables may have to be temporarily saved in memory and reloaded later. Use of this temporary storage is commonly called *spilling* and is generally recognized as a significant impediment to program performance.

Minimizing the impact of spills is one of the most important tasks of the register allocation pass. Spills affect execution time in a number of ways. Not only do they require additional instructions to store and reload the spilled values but these instructions may also interact with the slower memory system and take significant time to execute. However, simply reducing the number of spills emitted does not fully address the problem of their impact. Spills can increase code size but their execution time is also dependent on which control flow paths are followed during program execution. Therefore, attention must be paid to the dynamic behavior of a program as opposed to just its structure in order to better optimize spill cost reduction while improving program performance.

A good spill minimization strategy is often formulated in two parts: a heuristic for making spill choices during allocation of variables to registers and a spill instruction placement method that minimizes costs after allocation. It is difficult to do

both during allocation without significant computation cost. Spill choices are usually considered independently in some ordering of variables. The actual placement of instructions for a spilled variable can be dependent on how many places a variable is spilled within its range and on the spilling of other variables. Some of these decisions may not have been made yet so it is essential to have a good spill choice heuristic.

Register allocation algorithms rarely or only partially take the dynamic behavior of a program into account when spilling. Execution frequencies may be considered when making spill decisions but are usually combined to form a single spill cost estimate for the entire lifetime of the variable. While this estimate is easy to compute, costs can differ substantially over its lifetime.

A better spilling heuristic should consider the execution path instead of just execution frequency. Order of execution is important. Variables that are required soon again should be kept in registers while those with distant uses or are unlikely to be used again should be preferred for spilling. A well known heuristic that exhibits similar behavior is Bélády's unimplementable MIN algorithm. It chooses to spill the variable with the furthest next use [11]. This distance based heuristic implicitly relies on order of execution by considering where the next uses of variables are placed. Under certain conditions it can achieve the optimally minimal number of reloads because they are pushed far into the future.

The MIN algorithm is not without its problems. It was originally described in relation to an optimal virtual memory page eviction heuristic in which memory pages are considered in sequential order. Its application to program code, including conditional branching and loops, is not straightforward and was not covered by Bélády. When program code can branch to more than one location, multiple next uses are possible, each with their own distance. Determining a single distance to next use at the branch point is non-trivial. Spill costs, which may differ between variables, are also not considered by the MIN algorithm. Some may have to be temporarily stored to memory when spilled while others can simply be redefined by issuing the original instruction.

We describe and evaluate a register allocator that considers the dynamic behavior of a program in order to make spill decisions and optimize spill instruction placement. Our method of approximating distances applies Bélády's MIN algorithm to complex program control flow graphs by computing the Expected Distance to Next Use. While other attempts at simulating the MIN algorithm exist, we are the first to approximate distances to next use by using branching probabilities, and therefore to compute the

most accurate distances on average. We also describe solutions to the problems associated with the MIN algorithm when applied to program code including varying spill costs of variables and distances across edges where no use exists.

While our spill heuristic is surprisingly effective on its own, spill costs can be further reduced by optimizing the placement of the resulting spill instructions after allocation is complete. We present a near optimal spill placement method based on network flow problems and evaluate its effectiveness. It finds the least cost placement of spill instructions using only the dynamic execution counts across branches in the program code. Execution frequencies help to describe a dynamic representation of a program where the processor actually spends time executing instructions as opposed to the static structure of code.

1.1 Thesis Outline

This thesis describes a Distance based Register Allocator (DRA) and its evaluation. The thesis is ordered as follows:

Chapter 2 provides background on the topic of register allocation. It includes the context in which register allocation is placed, a description of the problems an allocator must solve, and a survey of the current state of register allocation methods.

Chapter 3 provides a description of our Distance based Register Allocator. This includes discussions on our method of allocation with spilling, move coalescing, and spill placement.

Chapter 4 provides an evaluation of our allocator with results. We compare to a popular state of the art allocator using static and dynamic profiling information and also compare to other heuristics and spill placement methods. We also provide a separate evaluation of our spill placement method.

We conclude the thesis in Chapter 5 by summarizing our contributions and indicating some items for future work.

Chapter 2

Background

Register allocation is an essential stage in a compiler for translating to target specific machine code. Its primary task is to assign physical registers, defined by the target architecture, to virtual registers representing program variables and intermediate (or temporary) results. The most significant problem faced by the allocator is due to the difference between the comparatively small number of physical registers and the potentially unlimited number of virtual registers. There may be points in a program where too many virtual registers are in use to fit into the available physical registers. When this happens memory must be used as a temporary storage area for "spilling," or storing virtual registers until they are needed again. Choosing which virtual registers to store then reload later must be carefully decided by the allocator as accessing memory can cause significant program delays.

The register allocator is a fairly complex compiler stage. Register allocation is not simply concerned with assigning registers but must also manage the placement of virtual registers over their lifetimes, both in physical registers and memory. The cost associated with allocation can be measured in the execution costs of additional store and load instructions due to spills, as well as register-to-register move instructions added to avoid spilling. Producing a good allocation of registers that minimizes these costs is difficult. It requires balancing solutions to several problems including avoidance of spills, making good spill choices when necessary, finding a low cost placement of the resulting spill instructions, and minimizing the number of move instructions required to manage the lifetime of variables in physical registers. Each of these problems must be approached with the overall goal of minimizing execution costs.

In this chapter, we outline the issues that make register allocation difficult. We

begin by describing the intermediate representation (IR) of the program being compiled. Its structure and organization pose a number of challenges for the allocator. We then outline the major problems that an optimizing allocator must solve and include a survey of different allocation methods. Given the complexity and importance of the register allocation problem it is no wonder that many methods have been proposed to handle the problem.

We also provide a short review of the literature on the complexity of register allocation. While register allocation, in general, appears to be quite complex, much work has been done on pinpointing where that complexity lies. Many claims have also been made on the optimality of solutions for particular stages. Often these claims assume a restricted version of the problem that cannot be adapted to the entire problem without losing optimality. We try to decode some of the claims in order to provide a more balanced view of the problem.

Register allocation also includes the subproblems of spill placement optimization and register-to-register move coalescing or elimination. Solving these problems is important for the performance boost they provide. Our overview on these problems discusses the different methods that have been proposed to solve them.

2.1 Intermediate Code

As was stated in the introduction, the typical programming language compiler is composed of three parts. The "front end" reads in the source code, validates it against the rules of the programming language used, and generates code in the form of a compiler specific *Intermediate Representation* (IR). The middle stage receives this IR code and performs analysis and optimization on it. Finally, the "back-end" receives the transformed IR code and generates target specific IR code with instructions that are identical, or close, to those of the target architecture. This machine IR is transformed by the back end to machine specific assembler code. The register allocation pass is part of the back-end as it requires specific knowledge of the target machine details.

The intermediate representation is based on an abstract machine that is neither specific to the input programming language nor to the target architecture. This independence not only simplifies the generation of the IR code but also of the analysis and optimization stages because the compiler does not have to manage target details which may be unrelated to the required analysis. Separation also supports reuse by allowing multiple input computer languages and output target architectures to use a

common platform for code generation.

IR code is commonly represented as a directed graph structure known as the *Control Flow Graph* (CFG) because it describes the possible execution paths in a program. Figure 2.2 shows the intermediate representation and control flow graph for the corresponding function written in C. *Basic Blocks*, such as `ForEntry`, are nodes in the graph that represent a maximal sequence of instructions with no branches into or out of the block except at its endpoints. A maximal sequence simply means that two blocks, connected by an edge, must be joined together if the predecessor has only one output edge and the successor has only one input edge.

Control flow enters at the top of a block and exits at the bottom. The number of successors of a block is dependent on the block's terminating instruction. For example, a conditional branch may have only two destinations. There are no limits on the number of predecessors that a basic block may have.

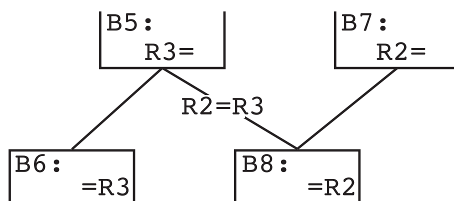


Figure 2.1: Splitting a live range with two definitions on a critical edge using a move instruction. Physical registers have already been assigned.

Some edges between blocks are termed *Critical Edges* because they must be split, and a new basic block inserted, if instructions are to be placed on these edges. An edge $P \rightarrow S$ is said to be critical if it is not the only edge exiting P and not the only edge entering S. Figure 2.1 shows a move instruction inserted onto a critical edge. Shifting the move instruction to the predecessor block would affect the definition of R2 in block B6. Shifting the move to the successor block would invalidate R2 if execution arrives at B8 from B7.

Basic blocks contain sequentially executed *Instructions*, similar to machine assembly code instructions. Each instruction may take a number of operands as input and produce others as output. The register allocation pass is concerned with *Virtual Register* operands. Virtual registers are analogous to physical registers in the target architecture except that there are no limitations, beyond the practical, on the number of virtual registers available. This potentially unlimited number of virtual registers simplifies the IR code by ensuring that it is not complicated by code for managing

```
// C source code
int findmax(int *data, int min) {
  int max = 0;
  for (int x = 0; data[x]; ++x) {
    if (data[x] > max)
      max = data[x];
  }
  if (max < min)
    max = min;
  return max;
}

// Intermediate Representation (IR)
int findmax(vreg0, vreg1)
;    vreg0 = int *data
;    vreg1 = int min
Entry:
    V0 = argument 0 ; data
    V1 = argument 1 ; min
    V2 = movi 0     ; int max = 0
    V3 = movi 0     ; int x = 0
ForEntry:
    V4 = add V0, V3 ; V2 = &data[x]
    V5 = load V4    ; V4 = *(&data[x])
    V6 = cmp V5, 0  ; for ( ; data[x]; )
    goto.eq V6<kill>, ForExit, IfCond
IfCond:
    V7 = cmp V5, V2 ; data[x] > max
    goto.gt V7<kill>, IfThen, ForBottom
IfThen:
    V2 = copy V5<kill> ; max = data[x]
ForBottom:
    V3 = add V3<kill>, 1 ; for ( ; ; ++x)
    goto ForEntry
ForExit:
    V8 = cmp V2, V1    ; max < min
    goto.lt V8<kill>, Update, Exit
Update:
    V2 = copy V1<kill> ; max = min
Exit:
    return V2          ; return max
```

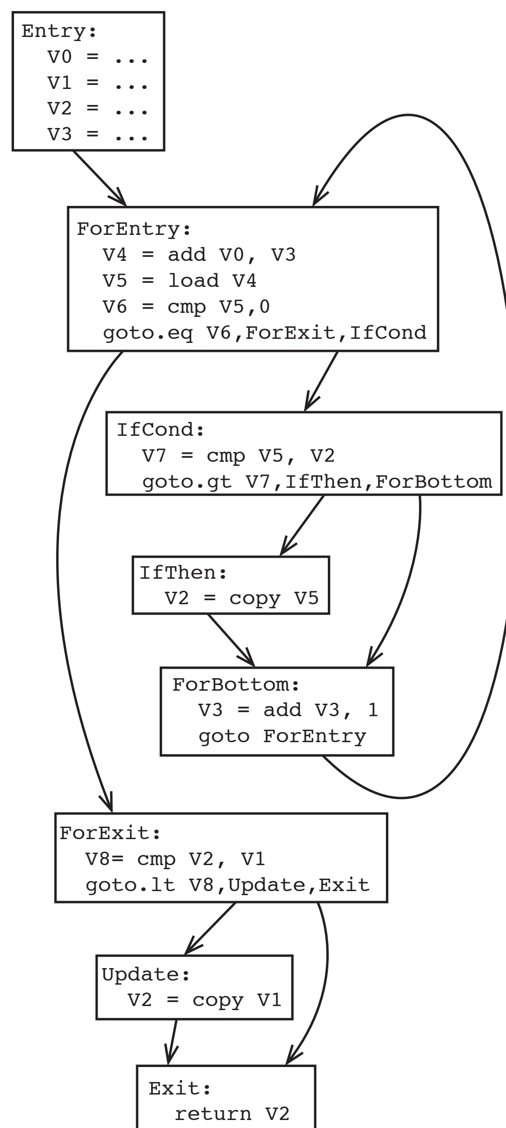


Figure 2.2: Intermediate representation for the function written in C, and its corresponding control flow graph

registers which would obscure the program logic.

The following IR instruction

```
V19 = V19 + 1
```

redefines the virtual register $V19$ by adding one to its value. The virtual register on the right-hand-side of the assignment operator is known as a *Use* of virtual register `V19` while on the left-hand-side it is known as a *Definition* of `V19`. Definitions always follow uses in execution order on a single instruction. Virtual register uses are evaluated first. The same instruction in annotated form is written as

```
V19<def> = V19<kill> + 1
```

Since the definition overwrites the previous value, the use of `V19` in this expression is said to be a *Kill* because there are no subsequent uses of the previous value. The *kill* marks the end of the lifetime of virtual register `V19` on the incoming path.

A register allocator will need to assign physical registers to the uses of virtual registers in an instruction before definitions. If a virtual register use is killed in an instruction then its assigned physical register will, generally, be available for assignment to a virtual register defined by the same instruction.

The *Live Range* of a virtual register is the region of the control flow graph where the virtual register is defined, or live. This range extends from each virtual register definition to the last use of the register on each control path. A virtual register is said to be 'live' at a given point in the flow graph if there is a path, moving forward, from that point in the graph to a use of the virtual register. This is important for a register allocator as it will have to ensure the virtual register remains defined over the lifetime of the live range whether it is in a register or stored in memory.

There are generally no limitations on the shape of the CFG, beyond those described for basic blocks. The flow graph for a function will normally have a single entry point but may have multiple exits. The shape of the flow graph for a function is largely dependent on the structures available in the input programming language. In the C programming language 'while' loops and 'if' statements allow for conditional branching. Many languages, including C and C#, also provide the "goto label" statement that allows for branching to arbitrary locations in the code. This can result in complicated flow graphs that can make analysis and optimization far more difficult.

2.1.1 Static Single Assignment Form

An alternative form of IR code is known as *Static Single Assignment* (SSA) form [34]. SSA is similar to the regular form except that it does not permit reassignment, or more than one definition for each live range. To support this property, a special instruction called a ϕ-function (phi function) is introduced that supports the merging of variables where control flow paths meet.

Figure 2.3 shows the IR code in SSA form for the function in Figure 2.2. To support the merging of multiple definitions along control flow paths, a special pseudo-instruction known as the ϕ-function (PHI function) was introduced. ϕ-functions are inserted where control flow paths meet and a virtual register is exposed to, or is reachable from, more than one definition of that virtual register. At these points the virtual register may contain different values depending on the control flow path taken to reach the ϕ-function. This ϕ-function takes as arguments the virtual registers corresponding to the different exposed definitions. For example, at the top of the `ForEntry` block, the variable `x` is exposed to definitions in the `Entry` and `ForBottom` blocks so a ϕ-function is inserted to merge these definitions. When inserting ϕ-functions into the code, virtual registers are renumbered to reflect the new, smaller, live ranges introduced.

The power of SSA form follows from the previous example. Many analyses require information about the definitions of a live range. For example, a register allocator can avoid storing a virtual register to memory when evicting it from a physical register if the virtual register represents a constant integer value. This information is contained in the definition. For the use of `x`, or `V10` in the `ForBottom` block of Figure 2.3, it is easy to determine that the definition resolves to a ϕ-function in block `ForEntry` and not a constant integer. Determining the same thing on the regular IR code is more difficult as a traversal of the control flow graph in the reverse direction may be required in order to find each definition of the virtual register.

The graphs of code in SSA form are known to have special structural properties that permit simpler and more efficient allocation methods [41, 59]. For this reason there has been a surge in interest in the area of register allocation on SSA form IR code.

Many programming languages, like C, do not possess the define once property of the SSA form, although the single assignment of functional languages has been shown to be similar in nature to SSA [3]. There are a number of methods for generating

```
// Intermediate Representation (IR)
int findmax(vreg0, vreg1)
;    vreg0 = int *data
;    vreg1 = int min
Entry:
    V0 = argument 0   ; data
    V1 = argument 1   ; min
    V2 = movi 0       ; int max0 = 0
    V3 = movi 0       ; int x0 = 0

ForEntry:
    V9 = phi(V2,V12)  ; max1
    V10 = phi(V3,V13) ; x1
    V4 = add V0, V10  ; V2 = &data[x1]
    V5 = load V4      ; V4 = *(&data[x1])
    V6 = cmp V5, 0    ; for ( ; data[x1]; )
    goto.eq V6<kill>, ForExit, IfCond

IfCond:
    V7 = cmp V5, V9   ; data[x1] > max1
    goto.gt V7<kill>, IfThen, ForBottom

IfThen:
    V11 = copy V5<kill> ; max2 = data[x1]

ForBottom:
    V12 = phi(V9,V11)      ; max3
    V13 = add V10<kill>, 1 ; x2 = x1 + 1
    goto ForEntry

ForExit:
    V8 = cmp V9, V1        ; max1 < min
    goto.lt V8<kill>, Update, Exit

Update:
    V14 = copy V1<kill>    ; max4 = min

Exit:
    V15 = phi(V9,V14)      ; max5
    return V15             ; return max5
```

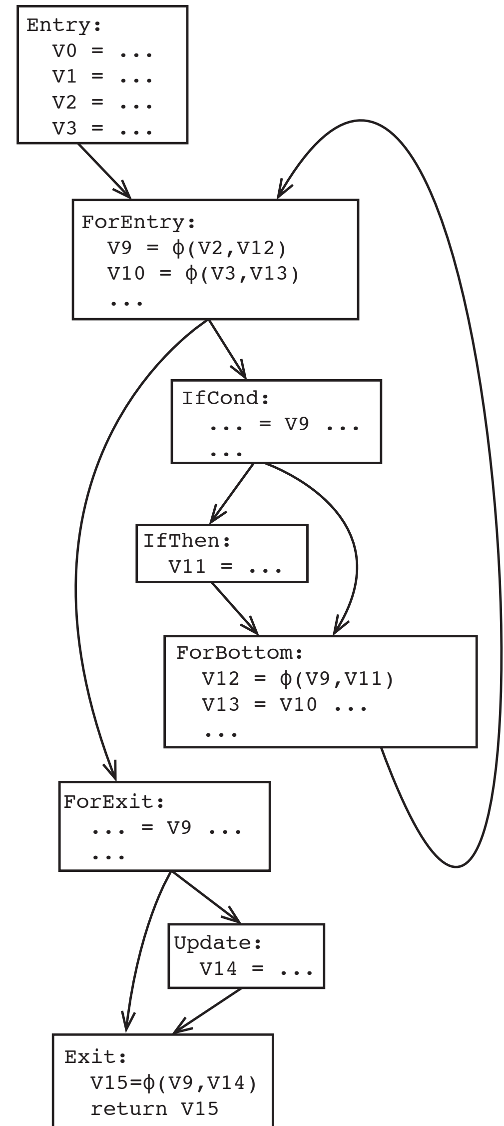


Figure 2.3: SSA intermediate representation for the C function of Figure 2.2, and its corresponding control flow graph showing only the renumbered virtual registers.

SSA form IR from the regular form input in polynomial time. The standard method, that is often cited when discussing the topic, can generate SSA form for arbitrary flow graphs [34]. However, if the flow graph is known to be structured — without goto statements — then generation of SSA form is somewhat simpler [19].

Since ϕ-functions are pseudo-instructions, and are not valid on any platform, SSA form must be translated back to the regular form in order to generate assembler code for the target architecture [66, 23]. ϕ-functions are easily eliminated using register-to-register move instructions. However, inserting move instructions without care can lead to inefficient code. Unfortunately, there is no known polynomial time translation from SSA form to regular form IR that produces optimal results for any input graph.

2.2 Register Allocation

The register allocation stage in a compiler is a complex piece of software. In order to produce efficient code, it must allocate a potentially large number of virtual registers into a small number of physical registers while trying to avoid spilling. Unfortunately, spills are not uncommon. Therefore, the register allocator must provide a means to reduce the costs of spills through the choice of what to spill and where to place the resulting spill instructions. Efficiency is also challenged by the managing of virtual register lifetimes. Issues such as merging control flow paths or predefined register placement can force the allocator to insert move instructions into the code. The allocator must be careful to minimize the number of move instructions inserted in order to produce efficient code.

The following list outlines the major tasks that a modern register allocator must perform. These tasks are not necessarily separate, although separation can lead to simpler implementation. Interactions between tasks can mean solving one could constrain the solution of another. Different allocation methods may combine or separate tasks depending on the algorithm.

- *Allocation* — Allocating registers is the process of deciding which virtual registers are in physical registers at each point in the input control flow graph.
- *Spilling* — Spill decisions may have to be made about which virtual registers to evict from a physical register when there are too many live virtual registers at a given point in the flow graph.

- *Spill Placement* — The placement of spills has an effect on program performance. Spills should be placed where they have the least impact on the running program.
- *Coalescing Moves* — Unnecessary move instructions can be eliminated by placing virtual register live ranges in the same physical register provided an unallocated register is available.
- *Assignment* — Assigning actual physical register numbers to allocated virtual registers.

Register Allocation is a loaded term in the literature. While it is commonly used to refer to a compiler stage it has also been used to refer to a decision problem of whether some given code can be allocated a specified number of registers without spilling [67]. This does not necessarily include the assignment of particular physical registers to virtuals. Virtual registers may simply be allocated to a register class or set of registers with the actual assignment decided later. Spill-free register allocation is an easier problem to solve than allocation with spills [67] because it does not involve deciding what to spill.

2.2.1 Spilling

Register spilling and the allocation of registers are tightly coupled. In fact, allocation cannot proceed if there are no unallocated physical registers available to hold a virtual register. The Spill Problem is the forced decision of which virtual register, currently held in a physical register, should be evicted to memory in order to make room for another virtual register where there are no other unallocated physical registers. Choosing which virtual register to spill is complicated by varying spill costs between virtual registers and interactions between spill decisions. Even the ordering in which virtual registers are considered can affect the resulting allocation. Actual spill costs may not be discernible until after allocation is complete.

Spills have traditionally, and for the most part still are, assumed to require temporary storage in main memory which is regarded as a very slow subsystem. This view is complicated by the varying spill costs between virtual registers where some may not require temporary storage to memory at all. In addition, the speed at which a value can be loaded from memory is difficult to determine when cache memory is available. If a value is in cache memory, then a load may take just a few clock cycles.

If the value is not in cache memory then it may take hundreds of clock cycles, or more, to load. However, even with cache memory it is not advisable to ignore store and reload costs. Too many spill instructions can increase the use of memory and therefore the probability of cache misses and significant execution delays.

Spilling can be forced for a number of reasons. If there is a point in the flow graph where the number of live virtual registers exceeds the number of physical registers available then spilling is inevitable. The number of available physical registers is not necessarily constant within a function. The set of physical registers available can become restricted as in the case of a calling convention where the contents of argument registers must be saved by the caller. Therefore, spilling can occur when the number of virtual registers is less than the number of allocatable physical registers.

If there is a point where the number of virtual registers is equal to the number of physical registers then spilling becomes less certain. Care must be taken to avoid the need to swap the contents of physical registers when all registers are in use as a spill may be required[1].

The cost of spilling a virtual register is largely based on the costs of the resulting store and load instructions inserted into the code. These costs are difficult to determine during allocation because they can be affected by spill decisions that have not been made yet. Each spill decision will affect the availability of registers and therefore the range that spill instructions could be moved.

Spill costs can also be complicated by the individual store and load costs of a virtual register. A virtual register is said to be *dirty* if the value it contains must be stored when spilled and reloaded from memory when next required. *Clean* virtual registers contain values that may be *rematerialized* or even recomputed without storing the original value [25]. Rematerializable instructions include copying a constant value such as zero into a register. Even loading from a fixed address, which is known at compile time, can be less expensive to spill because it avoids the need to store. Register allocators will tend to strongly prefer rematerializable values over non-rematerializable ones when making spill choices.

[1] Temporary storage in memory could be avoided using the three instruction XOR swap technique: `X = X xor Y; Y = X xor Y; X = X xor Y`. However, this can be expensive as the instructions must be executed sequentially and cannot be pipelined.

2.2.2 Spill Placement

Spill placement is the optimization of where spill instructions will be inserted into the code such that their execution costs will be minimized. Time spent on optimizing the placement of spill instructions is worthwhile given the potential costs of storing and loading to memory. The “Spill Everywhere” approach has been adopted by some allocators as a fast method of spill placement [28, 63]. It mandates the storing of a virtual register after each definition and reloading immediately prior to each next use. This is unlikely to be optimal as the spilled region of the virtual register, where it resides in memory, may only be a small part of its overall range. Spill everywhere can actually expand the region in which the virtual is in memory and lead to unnecessary spill instructions being inserted. However, this may reduce overall register pressure and prevent other virtual registers from spilling.

Discovering a better placement can be difficult due to the shape of the flow graph and interactions between live ranges. Spill instructions should be placed in less frequently executed points in the flow graph in order to minimize their execution count. Moving spill instructions outside of loops can significantly reduce costs. Moving spill instructions can affect the placement of others by using or freeing registers in a region. Even within a live range, moving a load instruction can affect the placement of the corresponding store, and vice versa, by increasing or reducing the range they must cover.

2.2.3 Move Coalescing

Move coalescing is the reduction of register-to-register move instructions by the allocator. While move instructions are inexpensive to execute, an excess of unnecessary moves can lead to code expansion, wasted clock cycles, and slower performance. Move instructions appear for a variety of reasons. A virtual register value may be in different locations on merging control paths such as in Figure 2.4 where a MOV instruction is required to place R4 into R3. Instructions may also have specific register requirements which force a virtual register to be placed in a specific physical register. Function calling conventions may also dictate where argument values are to be placed when function calls are made. In the figure, the calling convention expects arguments in registers R0 and R1. This forces the value in R0 to be moved to another register while the arguments in R4 and R2 are moved into the argument registers. Each of the moves shown in the figure could be eliminated.

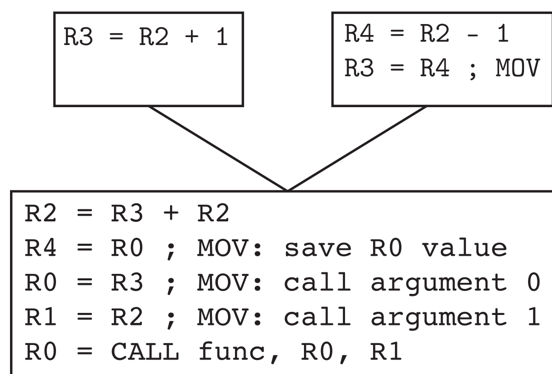


Figure 2.4: Unnecessary moves due to merging control flow paths and the function calling convention where R0 and R1 are argument registers and R5 is preserved across the function call.

2.2.4 Register Assignment

The assignment of physical registers to virtual registers may be performed concurrently with allocation or after it. If performed as a separate stage then the allocator will simply ensure that the set of registers that the virtual register may be assigned to, its *register class*, has an available physical register. The allocator does not need to consider if a specific register is available within the correct register class which means that move instructions will not be inserted during allocation. However, the assignment stage will need to consider where virtual register live ranges overlap and may need to insert move instructions to ensure that values are not accidentally overwritten.

Register assignment may also be performed concurrently with allocation. Virtual registers are assigned specific physical registers from the correct register class. Move instructions may have to be inserted during allocation to ensure a virtual register transitions from one physical register to another.

Architectural issues can complicate assigning registers due to constraints placed on an instruction's register operands. Some of these issues are outline in Section 2.2.6.

2.2.5 Calling Conventions

A calling convention is an agreed upon method for passing arguments to functions and returning results. Calling conventions are not defined by the hardware architecture but are instead mandated by the compiler to support interoperability between software components. The convention will define locations for passing arguments as well as which registers must be saved by the caller and callee functions.

It is generally more efficient to use registers than it is to pass them in memory due to the extra store and load instructions that would be required. However, there are a limited number of registers and a potentially unlimited number of arguments. A calling convention may choose to store the first N arguments in a fixed sequence of registers and place additional arguments in memory.

The Callee function, or function being called, uses the same set of registers as the Caller function. This means there is a potential for overwriting register values used by the caller. A calling convention will define which registers must be saved by the caller (caller-saved) and those that must be saved by the callee (callee-saved).

The calling convention poses a challenge for the register allocator because virtual register arguments must be placed in the correct argument registers prior to calling a function. This may involve inserting store, load, and move instructions in order to place virtual register arguments in the agreed upon argument registers. If the argument registers may be overwritten by the callee then the virtual register arguments may need to be saved prior to the call. This may mean copying them to a callee saved register or storing them to memory. Depending on the convention used, function calls and the mandating of caller-saved registers can represent one of the most significant reasons for spilling within a function.

2.2.6 Architectural Issues

Features of the target architecture can play a role in complicating register allocation. Processors are becoming increasingly more complex as features are added to improve performance. In order to realize performance gains the compiler must be able to exploit them or, at the very least, not work against them.

Instruction Pipelining Instruction pipelining can allow the processor to run instructions in parallel. Data hazards that interfere with pipelining can occur when register uses and definitions collide. For example, if an instruction defines a register immediately after one that reads the same register (write after read hazard) then the second instruction must delay until after the first instruction has finished with the register. These delays will slow execution speed.

Figure 2.5 shows a sequence of instructions for the ARM architecture for which registers have been assigned. The assignment prevents pipelining of instructions due to dependencies between registers. The first two move instructions assign to the lower

```
movw    r1, :lower16:addressA        movw    r1, :lower16:addressA
movt    r1, :upper16:addressA        movw    r2, :lower16:addressB
ldr     r2, [r1]                     movt    r1, :upper16:addressA
movw    r1, :lower16:addressB        movw    r2, :upper16:addressB
movt    r1, :upper16:addressB        ldr     r1, [r1]
ldr     r3, [r1]                     ldr     r2, [r2]
```

Figure 2.5: A register assignment that limits pipelining of instructions and another that does not.

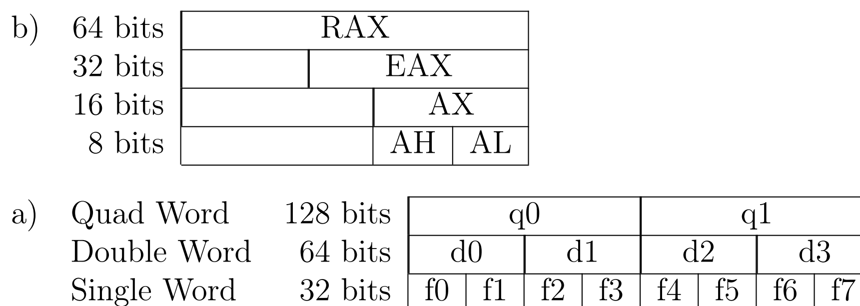


Figure 2.6: Register Aliases on the a) Intel x86, and b) ARM VFP architectures.

and upper halves of a 32-bit register and cannot be executed in parallel because the define the same register. The following load instruction must wait until the prior instructions complete so that it can load from the address that the moves define. If moves take one clock cycle to complete and loads take at least one cycle, then the first sequence of instructions will take at least six clock cycles to complete. The second sequence of instructions shows a different register assignment that supports pipelining. These instructions can be rescheduled to interleave them. Since these instructions can be pipelined, or executed in parallel, they can take as few as three cycles to complete.

Register Aliases Register aliases [68, 53] are names for registers that may partially or fully overlap. Assigning to one register name can affect the contents of the area referred to by another register name. Figure 2.6 shows two prominent examples of register aliases found in the Intel x86 and ARM floating point architectures where registers of different sizes overlap. When using the ARM VFP floating point registers, assigning a value to register d0 overwrites values in the single word registers f0 and f1. Aliasing causes problems when the value in one register blocks the use of another register.

Figure 2.7 shows a register allocation of floating point virtual registers that causes

a)

Floating Point Registers

Action	D0		D1		D2		D3		
	F0	F1	F2	F3	F4	F5	F6	F7	
define f0	f0	–	–	–	–	–	–	–	
define d0	f0	–	d0	d0	–	–	–	–	
define f1	f0	f1	d0	d0	–	–	–	–	
define f2	f0	f1	d0	d0	f2	–	–	–	
kill f1	f0	–	d0	d0	f2	–	–	–	
define d1	f0	–	d0	d0	f2	–	d1	d1	
define d2	f0	–	d2	d2	f2	–	d1	d1	move/spill d0

b)

Floating Point Registers

Action	D0		D1		D2		D3	
	F0	F1	F2	F3	F4	F5	F6	F7
define f0	f0	–	–	–	–	–	–	–
define d0	f0	–	d0	d0	–	–	–	–
define f1	f0	f1	d0	d0	f1	–	–	–
define f2	f0	f2	d0	d0	f1	–	–	–
kill f1	f0	f2	d0	d0	–	–	–	–
define d1	f0	f2	d0	d0	d1	d1	–	–
define d2	f0	f2	d0	d0	d1	d1	d2	d2

Figure 2.7: Aliased register allocation showing the potential for interferences between registers of different sizes to cause spills. a) A poor allocation causing a spill, b) A successful allocation. "define f0" places the virtual register f0 into a physical register. "kill f1" frees the physical register of virtual register f1.

an unnecessary spill. The register set is similar to, but smaller in size, than the ARM VFP register set. In the example, both 32-bit (f) and 64-bit (d) virtual registers are placed in 32-bit (F) and 64-bit (D) physical registers. At each step an action is taken by the allocator: a virtual register is defined into a physical register of the corresponding size, or a virtual register is killed and the physical register deallocated. In example a), a virtual register is spilled or moved because there is no unallocated 64-bit physical register available despite there being two 32-bit physical registers available. Example b) shows a better allocation of registers that avoids this problem.

Register Pairs A similar feature to register aliasing is register pairs [24]. On some architectures, instructions may define or use a pair of adjacent registers as arguments. For example, the ARM architecture defines the `STRD` and `LDRD` instructions for storing and loading multiple values. They can be used to manipulate two values in one

instruction and may be faster than using two single register instructions. However, they can complicate register allocation as the register pairs must be in adjacent registers. This can force move instructions to be inserted or even spill instructions to ensure the correct assignment of physical registers. Existing allocation methods may need to be modified to handle register pairs [24].

2.3 Register Allocation Methods

Register allocation has received considerable attention given its importance as a necessary compiler stage. A number of methods and optimizations of these methods have been proposed. Given the complexity of the problem and the wide range of compiler applications it is hardly surprising that register allocation has received such a high level of interest.

2.3.1 Local Register Allocation

Local register allocation is the allocation of registers in straight-line code. Execution frequency is not used to determine spill costs since all instructions share the same execution count. This means that the placement of spill instructions is not generally a concern, although specific architectural issues may affect placement. Stores can be placed after definitions and loads immediately prior to uses.

Horwitz et al. [42] developed an optimal register allocation method using dynamic programming for processors with index registers. Hsu et al. [43] followed Horwitz's method with their own modifications for code without index registers. They create a weighted directed acyclic graph (WDAG) where nodes represent the possibilities for assignment of variables to registers at an instruction. Their solution is found by finding a shortest path through the WDAG. Since there may be several possible configurations the possibilities can grow exponentially. They must prune the tree to keep their approach feasible. Their algorithm is therefore, no longer optimal.

In 1966, Belady [12] showed that when page evictions are required in a memory paging system the optimal choice is the page with the furthest next use. This will result in the minimum number of page loads. Unfortunately, the exact sequence of page reads isn't known in advance. This means that a prior run of the program would be needed in order to learn this sequence in order to achieve optimal performance on the second run.

While Belady's MIN heuristic is impractical for memory paging, it appears to be a good fit to register allocation as the exact sequence of instructions for straight-line code (and can be approximated for control flow graphs) is known at compile time. In fact, some time earlier in 1955, Sheldon Best developed and implemented a register allocator that used a furthest next use heuristic in a FORTRAN compiler [8], although this information was not published at the time.

Unfortunately, the MIN algorithm is designed only to minimize page loads while assuming all pages have equal costs to write-back to memory. Pages, and registers, can be "dirty" or "clean" which dictates whether the page needs to be stored when evicted. Dirty values must be stored when evicted as their contents has been modified. This will incur additional costs beyond that of reloading. The MIN algorithm offers no guidance on choosing dirty versus clean values when evictions are forced.

Approximations of the MIN algorithm have been proposed such as the *Conservative Furthest First* heuristic [36, 55]. Given the set of variables with the furthest next use, consider one for eviction in the following order: evict one that is not live, evict one that is clean, choose an arbitrary dirty variable for eviction. This heuristic yields a good approximation of the optimal solution. Another heuristic is the *Clean First* (CF) heuristic which will choose to spill the clean variable with the furthest next use even if its distance is much closer than a dirty variable. This heuristic hopes to balance store and load costs by reducing stores at the expense of a potential increase in rematerialized instructions.

The MIN algorithm has been applied to allocation of registers in long basic blocks with good results [40] where both the MIN algorithm and a CF heuristic were evaluated. The authors note improvement over a graph coloring allocator on blocks with lengths from hundreds to thousands of instructions. However, for some code the CF heuristic tended to spill registers that were needed soon again which would cause an increase in spill instructions and a decrease in program speed.

2.3.2 Graph Coloring

Graph coloring of virtual register *interference* graphs was proposed as an allocation method by Chaitin et al. [29, 28]. Graph coloring is the problem of assigning colors to nodes in a graph such that no two nodes that are connected by an edge share the same color. A minimum coloring of the graph uses the least number of colors while a K coloring uses at most K colors.

They noted the similarity of graph coloring to register allocation where virtual registers that are live at the same time cannot be placed in the same register, or assigned the same color. They defined the interference graph where nodes in the graph represent virtual register definitions and uses. An interference edge is connected between interfering nodes and represents the constraint that they may not share the same physical register. A solution to the allocation problem is a coloring of the virtual register nodes using a set of K physical registers.

Some graphs are not K-colorable. More than K interfering virtual registers may be live at the same time. When this happens, some virtual register is chosen to be spilled. Chaitin's spilling heuristic was based on a *Spill Everywhere* approach where the chosen virtual register would be placed in memory over its lifetime. A store instruction would be inserted after each definition and a load prior to each use. The spill decision is therefore driven by a cost estimate based on the execution costs of the required store and load instructions [28].

Register-to-register move instructions are also addressed by the graph coloring allocator. Move related edges are added to the graph between nodes that are connected by a move instruction. As originally proposed, an aggressive coalescing approach was used to reduce these edges. If two nodes are connected by a move related edge and those nodes do not interfere then they are coalesced into the same node. Understandably, aggressive coalescing is very successful at reducing move instructions. However, it can lead to an increase in nodes with high degree $> K$, a graph that is no longer K-colorable, and an increase in the number of spill instructions inserted.

Briggs et al. [22] introduced some improvements to Chaitin's algorithm. They add a conservative coalescing stage that only merges move related nodes if the resulting node has degree $\leq K$. The graph will not become uncolorable after conservative coalescing because the new nodes are K colorable due to their low degree. They also add an "optimistic" approach to coloring whereby they remove a spilled node from the graph but do not label it as spilled in the hopes that it may be colorable later.

Further improvements to the graph coloring allocation method were proposed by George and Appel [38]. Their algorithm forms the basis for many current graph coloring based allocators. Figure 2.8 shows the block diagram for this algorithm. After building the interference graph, the Simplify stage removes non-move related nodes of low degree $\leq K$. These nodes will always have a register available no matter what register is assigned to their neighbors. By removing them, other nodes may be simplified. When Simplify fails, the Coalesce stage applies Briggs' conservative

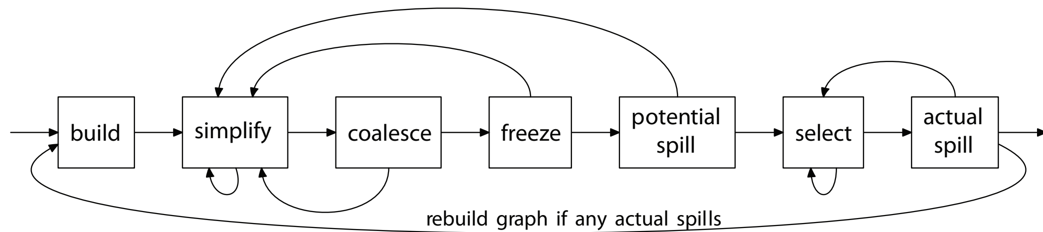


Figure 2.8: The stages of the Iterated Register Coalescing Graph Coloring allocator[38] as shown in [5].

coalescing. Neither of these stages will make the graph non-K-colorable. Iterating between these two stages can potentially remove more nodes from the graph than either stage alone.

When Simplify and Coalesce fail, the Freeze stage chooses a move related node of low degree and labels it as non-move related. This allows the node to be coalesced or simplified at the cost of potentially accepting the move instructions. Their Potential Spill stage relates to the Briggs' "optimistic" coloring approach. A spilled node is removed from the graph but may be revived later when registers are assigned to the graph nodes.

While graph coloring is one of the most popular allocation methods it is not without problems. It has been pointed out that graph coloring is not the same problem as register allocation [67] as there are graphs for which the coloring approach will not find the best solution without modifications. The graph coloring approach simply models interference relationships between virtual registers. Additional modifications are required to handle architectural issues such as register aliases and overlapping register classes [68].

Virtual register spill decisions are based on static cost information about definition and use placements that is combined into a single metric. Accuracy can be improved using dynamic profiling data but the algorithm remains the same. Information about order of execution, or proximity between register uses, or even number of uses is lost during construction of the interference graph. For this reason, local register allocation has been found to be superior to graph coloring within basic blocks [43, 40].

Graph coloring allocation tends to be expensive to compute. They require that an interference graph be built and rebuilt whenever spills occur. There are efficient ways to build the graph [33] but the graph coloring allocation method is still regarded as inappropriate when compilation speed is important. Cooper et al. [32] designed a graph coloring allocator that builds the graph once. By not rebuilding the graph

when spilling occurs, errors are introduced. They still find that the solutions are only slightly worse than the accepted method.

2.3.3 Linear Scan

Linear Scan Register Allocation is a technique that was designed for very fast allocation speed at the expense of efficiency of the resulting code [62, 63, 72]. It is well suited for dynamic or Just-In-Time compilation [6].

The basic technique involves linearizing the control flow graph into a single sequence of basic blocks. This sequence also represents a list of sequentially numbered instructions. Each virtual register lifetime is represented by a *Live Interval* over the linearized graph. In the simplest form, an interval can be represented by two numbers. The first is the lowest index of the definitions for the interval while the second number is the highest index of all the uses of the virtual register. Later implementations used a more sophisticated approach where a live interval is represented by an ordered list of subintervals that more accurately represents the regions over which the interval is defined and where it is not.

Poletto and Sarkar's [63] original linear scan method begins by sorting the intervals by increasing start number. The list is then scanned from first interval to last with each interval assigned an available physical register. Checking if two intervals overlap, or are live at the same time, is simply a matter of detecting overlap between start and end points. If two intervals overlap then they cannot be placed in the same register. If all registers are in use and a new interval is defined then they choose to spill the interval that has the furthest end point.

One of the problems with this method is that there may be "holes" within an interval where the virtual register is not live. Two overlapping intervals may not actually interfere if one interval can fit within the holes of another. Traub et al. [72] addressed this problem using a bin packing method. Their intervals contain subintervals over the instructions where a virtual register is live. This allows them a much finer level of control when checking where intervals overlap. They also support splitting of live intervals which allows a virtual register to be placed in different registers over its lifetime.

Mössenböck and Pfeiffer applied linear scan to programs in SSA form [57]. However, during interval building they convert to non-SSA form. While they do not support splitting of live intervals, they note that intervals tend to be smaller in SSA

form when intervals are split due to the introduction of ϕ-function instructions. Wimmer and Mössenböck improve on this work by supporting splitting of intervals [74]. They also add spill cost reduction techniques including moving spill points outside of loops, reducing spill stores, and removing unnecessary register moves.

Sarkar and Barik describe an Extended Linear scan (ELS) algorithm that uses graph coloring spill cost estimates when making spill decisions [67]. They contend that their ELS method is far more efficient than graph coloring but can produce code that is just as competitive. However, the scope of their comparison is limited to spill free allocation and allocation with total spills where a spilled virtual register is only accessed directly from memory. They do not consider optimizing the placement of spills.

Wimmer and Franz present a linear scan algorithm that works on SSA form directly [73]. They do not translate to regular form IR prior to allocation. They use SSA properties to simplify the allocation process. By carefully ordering blocks they guarantee that definitions are prior to uses for any given virtual register. This allows them to eliminate the data flow analysis pass for recognizing live intervals. SSA form is deconstructed as part of the final allocation phase.

2.3.4 Other Methods

A number of other global allocation designs exist apart from the most well known.

Koes and Goldstein [47] describe a "progressive allocator" that finds an initial allocation that it can then improve upon given more time. They relate allocation to the Multi-Commodity Network Flow problem. Operands are commodities in the flow network that are assigned costs both to place in a register and into memory. Computing a solution is simply a matter of finding the shortest path through the network. Computing better solutions means taking into account how the allocation of a single variable affects others which requires more time to compute.

Pereira and Palsberg [65] transform the register allocation problem into a puzzle where program variables are puzzle pieces and the register file is the puzzle board. They achieve a polynomial time allocation algorithm while using Bélády's furthest first heuristic for making spill choices. They do not give preference to any path for determining distances and expect that this could be improved using profiling information.

Integer linear programming (ILP) is a powerful method for solving difficult prob-

lems. It can be used to model the register allocation problem in great detail by constructing a set of linear equations. Variables in these equations represent different actions performed during allocation. These equations can be used to model all aspects of allocation including irregularities in the instruction and register sets. Once the problem has been described the equations are solved by finding a minimum cost assignment to the variables. ILP allocation is well suited to solving allocation for irregular architectures due to its expressive power [4, 50]. While ILP allocators can produce near optimal results, they tend to take much longer to solve than conventional methods.

2.4 Complexity Of Register Allocation

Optimal register allocation is difficult. Modern allocators must solve several problems with the overall goal of minimizing the impact of the solution on running code. This impact is primarily due to the cost of extra instructions inserted by the allocator to manage the lifetime of virtual registers over their lifetimes.

It is not clear how to best solve the register allocation problem. The stages of register allocation, including allocation, spilling, spill placement, and move coalescing, can be combined or separated depending on the method. Each problem has its own complexities and interactions with others. While intuition suggests that register allocation is an NP-hard problem it is not clear what that hardness can be attributed to.

Graph coloring forms the basis of one of the most popular register allocation methods [29, 28]. Register operands become nodes in an interference graph where two nodes that are live at the same time are connected by an interference edge. The object is to color nodes with no more than some number K of colors such that no two connected nodes are assigned the same color. This problem is known to be NP-complete for arbitrary graphs when $K > 2$ [37]. Chaitin et al. [29] recognized that the NP-completeness of the graph coloring problem on arbitrary graphs lead to the NP completeness of their register allocation algorithm. While this could lead to impractical running times on some inputs, their experience showed that this problem was not a significant concern.

One possible solution to reducing the complexity of register allocation is to restrict the shape of the input graphs to those that are easily colorable. Thorup showed that structured programs, with conditional and loop statements but no 'goto label'

statement, have graphs with small tree-width [71]. This property allows them to be colored within a small factor of optimality in linear time. So, if the input language can be constrained to produce only certain types of graphs then the compiler may be simplified and may be able to color more graphs in K colors with fewer spills.

A different approach to restricting the shape of the input graphs is to transform the input into SSA form. In SSA form there is only one definition point for any virtual register. A virtual register live range may be broken into several smaller live ranges each with only a single definition point. This leads to interesting properties that allow coloring in polynomial time of arbitrary graphs. Hack and Goos [41] show that programs in SSA form can be colored optimally in polynomial time provided a K coloring exists. Their discovery stems from the fact that programs in SSA form have interference graphs that are *chordal*. Chordal graphs have a perfect elimination ordering which can be used to find the maximum chromatic number and an optimal coloring of the graph in polynomial time. However, this optimality does not extend to the different problem of spilling. If the graph cannot be colored in K registers then these properties do not offer a solution to optimal spilling.

The biggest problem with allocation on SSA form and its promise of optimality is that translation out of SSA is hard. ϕ-functions, used for merging distinct paths of a variable, are not a valid instruction in any assembler instruction set. Programs must be converted into SSA form, processed by the allocator, then converted out of SSA form once allocation or any other optimizations are complete. Fortunately, polynomial time algorithms exist for transforming into SSA [34, 7, 19]. However, polynomial time algorithms for transforming out of SSA form with the optimally minimum number of move instructions do not exist unless P=NP. The problem lies with the elimination of ϕ-functions which are replaced with copy (or move) instructions. It is the coalescing and placement of the move instructions, such that their costs are minimized, which makes the problem NP-hard [66].

Pereira and Palsberg [61] have also revisited Chaitin's proof on the NP-completeness of register allocation. They offer a proof that transforming out of SSA form, with the optimally minimum number of copy instructions, after register allocation in polynomial time is not possible unless P=NP. So, while SSA may offer a path to allocating graphs that can be colored in K registers, it has not simplified the overall problem of register allocation.

Bouchez et al. [16] revisited Chaitin et al.'s proof in light of the discoveries of polynomial time register allocation on SSA form. They conclude that the NP-completeness

comes from the presence of *critical edges*, optimization of spill costs, and the optimization of coalescing costs. If edge splitting, with the insertion of a new basic block on an edge, is not allowed then it is difficult to determine where to split live ranges with critical edges. Claims of optimality when allocating on SSA form do not include choosing spills when graphs cannot be covered in K registers. Minimizing the costs of spills remains an NP-complete problem.

While move coalescing is an important part of register allocation it has, perhaps, garnered increased interest as a part of the transition out of SSA form. ϕ-functions are replaced by move instructions which can then be coalesced. Bouchez et al. [17] analyze several coalescing heuristics and show their NP-completeness. Claims of the optimality of register allocation on SSA form must be tempered by the NP-completeness of the out-of-SSA transition and move coalescing.

Local register allocation differs from global allocation because the code executes sequentially without branching and therefore, all instructions have the same execution count. One might conclude that local register allocation is a much simpler problem. However, Farach and Liberatore [36] found that even performing register allocation on straight-line code can be NP-hard. They found that optimal allocation is possible if only the reload costs are considered. These can be minimized using a *furthest first* spilling heuristic [12, 43]. However, if both store and load costs are considered the problem becomes NP-hard. This is due to the fact that load costs can vary depending on how many times a virtual register is spilled, store costs are actually a binary decision. The virtual register is either stored or it is not and the cost of storing is only paid once, no matter how many times it is spilled.

2.5 Spill Code Placement

Spill decisions and spill placement are interrelated problems. The minimization of spill costs is the primary focus of register allocation and both problems are aimed at doing just that. Spill costs are primarily based on the types of instructions required to maintain a virtual register over its lifetime, and where these instructions are placed. Decisions on what to spill during allocation are largely based on minimizing these costs and therefore, assume a placement of spill instructions.

Finding the best placement of spill instructions can be difficult during allocation if choices are made incrementally. The most common allocation methods, including graph coloring and linear scan, are incremental in nature because the heuristics used

for deciding on what to spill apply to virtual registers independently. The optimal spill choice among a selection of candidate virtual registers may not lead to an optimal allocation of registers over the entire function.

Spill choices can affect future spill decisions. When a virtual register is spilled it is placed in memory over some region of the control flow graph. Register pressure will be reduced over this region which can, in turn, reduce the need to spill other virtual registers.

"Spill everywhere" placement is a simple approach used by early graph coloring allocators [28]. Costs are straightforward to compute as the sum of a store instruction after each definition and a load prior to each use. It has the desirable effect of reducing the number of virtual registers that are live over the graph and may help to avoid spilling of other virtual registers. However, it can also lead to higher spills costs than is necessary as the region where register pressure is high may be very small and require far fewer spill instructions.

More sophisticated spill placements are possible. If an unallocated physical register exists near a spill then parts of a live range may be revived. Spill everywhere may insert multiple reloads in a single basic block for each virtual register use. If an unallocated register exists then some of these reloads may be removed by reviving the virtual between reloads [15]. Code motion can also be used to revive parts of a live range between basic blocks. Koseki et al. [51] advocate keeping track of unallocated registers in order to move loads backwards in the flow graph towards store instructions. When a load meets a store the load, and possibly the store, can be eliminated, although a move instruction may be required.

Example 2.5.1. Figure 2.9 shows a virtual register live range with two definitions (D) and a single use (U). The allocator has been forced to spill (S) inside of a loop and reload (L) the virtual register on the loop back edge. It has also been forced to reload on the loop back edge as the live range is live into the block. There is also a load on the edge into the block with the use. The cost of spilling the virtual register is 496 store instructions executed and 496 loads executed. Assuming a spill everywhere placement the costs to store would be 64 as there are two definitions and the cost to load would also be 64. The least cost placement of spill instructions is actually 16 stores and 16 loads since spilling only occurs on the left-most path of the live range.

Modifications or additions to the interference graph have also been investigated in order to find a reduction in spill instruction costs. Kolte and Harrold [49] break

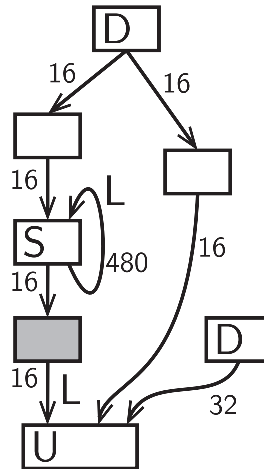


Figure 2.9: A spilled virtual register live range.

live ranges into smaller "Load/Store" ranges and perform graph coloring allocation on these. A load range extends from a definition or a use to the next uses. A store range extends from a definition to all uses it covers. These smaller ranges are used to construct the interference graph rather than the entire live range. When spilling a node in the graph spill instructions are inserted in the load/store range rather than the entire live range.

Bergner et al. [14] introduced interference region spilling to limit the area in which a live range is spilled. The interference region corresponds to the area in which two live ranges interfere. When spilling one of the live ranges, stores are placed after each definition that can reach the interference region and before each use in the interference region. Their heuristic avoids much of the unnecessary spill code of Chaitin's allocator.

Callahan and Koblenz[27] overlay the CFG with 'tiles' that contain a portion of the graph such as a block, a loop, or other tiles. These tiles, which cannot partially overlap, form a tree on the flow graph. Spills that were decided in a tile can be moved to less frequently executed regions of the graph by moving their spill code up the tile tree. Cooper et al. [31] compared the Callahan Koblenz allocator with the Briggs allocator and found significant improvement over the spill everywhere approach used in the Briggs allocator. Being able to split a single live range and allocate the parts independently proved to be very helpful.

Separation of spill choices and spill placement can allow for a better optimization of spill costs on the set of chosen spilled virtual registers. Once allocation is done, the

allocator has complete information on what is spilled and where. It can then refine the placements of spill instructions without the set of spilled virtual registers changing. However, separation is not necessarily better as the improvements a placement pass can make will be limited by the quality of spill choices made. Approaches based on integer linear programming can evaluate the problem as a whole to find the best selection of virtual registers to spill that minimizes the resulting spill instruction costs. These approaches are expensive to compute and often rely on heuristics to reduce complexity, and consequently optimality.

Appel and George [4] break the register allocation problem into the sub-problems of spill code placement followed by coalescing of registers. Their integer linear programming model was used to find an optimal solution to the spilling problem prior to assigning physical registers. However, their spilling was only optimal based on their model which was not able to consider that a variable could be in memory and a register at the same time. This restriction means that variables cannot be pre-stored such as at the definition of a live range where execution costs may be very low. Their approach does suggest that separation of the problem into "two phases does not significantly worsen the overall quality of the solution."

Proebsting and Fischer [64] separate allocation from register assignment in their "Demand Driven" allocator. They perform local register allocation and spilling on each basic block prior to a global allocation pass. They claim this separation simplifies the allocator and improves the allocation results. Spills are determined through a probabilistic approach where the likelihood of a variable staying in a register is inversely proportional to the distance to next use. The global allocation pass attempts to remove loads and stores in the graph or at least move them outside of loops.

Ebner et al. [35] provide a separate aggressive spill placement optimization approach. They apply the Minimum Cut problem to solving the placement of loads in spilled live ranges. The Minimum Cut of a graph is a separation of the nodes into two sets. The edges, from the original graph, connecting the nodes between the two sets form a cut of the graph. The cut is minimum when the sum of costs of these edges is minimal across all possible cuts of the graph. When applied to spill analysis, the graph is a subregion of the control flow graph where spill instructions may be placed. The minimum cut of this restricted graph will form a minimum cost placement of spill instructions for the spilled virtual register. They combine all of the minimum cut problems for each spilled live range into a single network flow problem which they call the Constrained Minimum Cut problem. They solve this Minimum Cut problem

using integer linear programming. Combining these problems allows them to consider register pressure and other interactions between live ranges. However, they do not optimize store placements in the graph, although store costs are included in the model.

Linear Scan register allocation gained initial popularity as an allocation method primarily due to its speed. However, with increases in processor speed and a desire to produce more efficient code, improvements have been made to the quality of code that it can produce. As originally proposed by Poletto and Sarkar, Linear Scan used the "Spill Everywhere" approach on a spilled interval [63]. Traub et al. introduce interval splitting in their linear scan allocator which allowed an interval to transition between different locations [72]. Spills could then be placed in a subsection of the original interval. Wimmer and Mössenböck [74] proposed searching for optimal split positions in order to minimize the execution count of stores and loads. They consider the split point as an upper bound on the execution cost of a spill and try to move the spill instruction outside of loops or to prior block boundaries which may render them unnecessary.

2.6 Move Instruction Coalescing

Register to register move instructions are one of the least expensive instructions on any given architecture. However, unnecessary move instructions can degrade code quality as they contribute to code expansion and an increase in execution time. The larger the code base the more memory required to hold it. For processors with cache memory, excess move instructions can increase the number of cache misses during execution which can cause significant slowdowns. Allocators will try to minimize the number of moves that they insert whether it is during allocation or as a separate pass over the code.

Moves may be inserted by an allocator for a number of reasons which can depend on the allocation method used. A local allocator may insert a move within a basic block if a virtual register is forced out of a physical register. It would be preferable to move the virtual to an unallocated physical register rather than spill it to memory if that is an option. Calling conventions can prompt this sort of situation. If the calling convention specifies argument registers then a virtual register may have to be moved into one prior to the call, or it may have to be moved out of one in order to be preserved across the call. It is also possible to position a virtual register in two

locations so that it is preserved across a function call but is also passed to the call as an argument. An allocator may also define edge moves in order to reposition a virtual into a register location at the top of a successor block. All edges into the successor block must be in the same location whether that is a particular physical register or memory.

Coalescing virtual register live ranges merges those live ranges into the same physical register location. Since the goal is to eliminate move instructions, merging live ranges that are not connected by a move is most likely not profitable and could make the problem harder. Also, some moves cannot be coalesced because their live ranges interfere.

Move coalescing requires information on, or the ability to check for, interferences between live ranges. A live range cannot be placed in a new register if it is already inhabited by another virtual register at some point in its lifetime. An interference graph provides a good representation of the problem. Linear scan allocators can check for interferences through interval overlap checking.

Coalescing has been widely studied in graph coloring allocators where it is combined with allocation [29, 28, 22, 38]. Move-related edges are introduced into the graph between nodes that are joined by a move instruction. These edges are aggressively eliminated by continuously merging move-related nodes that do not interfere with each other. This approach can lead to an increase of nodes with high degree, making it harder to color the graph, and most likely increase the number of spills. Interestingly, for intermediate code in SSA form, going out of SSA form prior to register allocation has been likened to aggressive coalescing [17] because it ignores the potential effects on spilling.

Reducing the number of move instructions is important but it is not beneficial to do so at the expense of introducing more spills to memory. Briggs et al [22] offered a conservative coalescing heuristic that would only merge along move-related edges if the resulting node was still colorable. This approach was borrowed by George and Appel [38] who exclusively use the conservative coalescing heuristic. They combined conservative coalescing with graph simplification that removes non-move related nodes from the graph that are of low degree. Iteration between these two heuristics allowed them to achieve a greater reduction in move instructions than the conservative heuristic alone.

Park and Moon [60] suggest that the conservative approach is too conservative because it does not consider the effects of merging two nodes on their neighbors. If

a node x interferes with both nodes being merged along a move-related edge then the degree of x will be reduced by one. This may make x colorable even if the new merged node becomes uncolorable. They proposed an *Optimistic* heuristic that would aggressively coalesce along move-related edges while allowing for nodes to be uncoalesced, should it be required at a later time, in order to avoid spills.

Bouchez et al. [18] found that the ordering of nodes is important. Firstly, nodes should be ordered by decreasing weight so that the strongest, or most heavily executed, are considered first. This is commonly applied in most methods. However, they also found that the ordering of nodes of equal weight can also affect results for the conservative heuristic. In fact, using this extra information they found the conservative heuristic of Appel and George outperformed the optimistic heuristic of Park and Moon.

Bouchez at al. [18] have also offered a different take on the conservative coalescing heuristic where the aim is to maintain the k-colorability of the graph. They note that coalescing that results in a node of high degree does not necessarily lead to an uncolorable graph. In fact, some graphs may have coalescable nodes that do not meet the simplification and coalescing criteria of George and Appel. Bouchez et al. present a Chordal-Based Incremental Coalescing algorithm that seeks to find a "path" between two nodes x and y of non-interfering nodes. The nodes x and y can only be coalesced, regardless of the resultant degree, if such a path exists. If a path does exist then each node on the path can be coalesced together.

Interference graphs are expensive to build and should be avoided when compilation speed is important. A separate analysis pass can be used to determine *preferred* physical registers for virtual register live ranges. These register preferences can then help to guide allocation and reduce register-to-register move instructions without the need for an interference graph or separate coalescing pass [21]. Linear scan allocation requires recognition of lifetime holes and interval splitting to perform move coalescing [63]. Register hints have been used in linear scan allocators [74] where the source interval for a move provides a register hint to the destination.

It is also worth noting that extensive analysis may not be necessary for good results. Koes and Goldstein [48] found that a simple greedy heuristic was sufficient to provide good quality code.

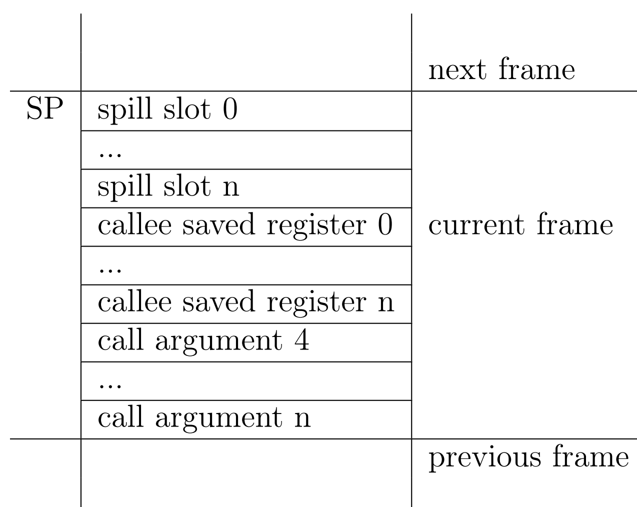


Figure 2.10: Example stack frame layout for the ARM architecture. The first four arguments are held in R0-R3. The function return address is held in the LR (R14) register. The SP (R13) register holds the stack pointer which points to spill slot 0.

2.7 Beyond Register Allocation

What we've presented here is an overview of the major tasks of a register allocator in an optimizing compiler. Often this is the extent of the discussion on register allocation. Yet the artifacts that the allocator produces, spill or move instructions and their placement, have effects on the intermediate code beyond the allocation pass. Ignoring these effects can have a negative impact on program speed.

2.8 Spill slots and the data cache

Spilled virtual registers are stored in memory on the function call stack as in Figure 2.10. This area is allocated when program execution enters the function and is deallocated when the function is exited. The assignment of *slots* in the call stack is a post-register allocation task yet this assignment is made necessary by the register allocation pass. The assignment of stack slots is optimizable and can have a impact on program speed.

The assignment of stack slots is important due to the effects of cache memory. If a read or write to the stack misses the top level cache then there could be significant delays. A memory location is kept live in the cache if it is accessed frequently enough in relation to other data that would share the same location in the cache. Since cache

memory is organized in lines, where a line represents a contiguous sequence of bytes in main memory, the accessing of several memory locations will contribute to the cache line remaining live in the cache. This leads to optimization of the assignment of stack slots.

Spills should inhabit as little space as possible on the stack. It is only necessary to assign unique stack slots to spilled virtual registers if they are live at the same time. If two spilled virtual registers are never in memory at the same time then they may be assigned the same stack slot.

Stack slots for spills should be assigned so that cache misses are reduced. Sharing slots helps to reduce the size of the stack slot area but slots can also be ordered so that cache lines are more likely to remain in the top level of the cache. One method, employed by LLVM [52], is to group slots by access frequency. Access frequency for a virtual register can be determined by adding the execution cost of each of its spill instructions. Execution costs can be determined by estimate or through dynamic profiling. Stack slots can then be assigned to each spilled virtual register in descending access frequency order.

Another possibility is to use the order in which virtual register spill instructions are executed to determine stack slot adjacencies [54]. If virtual register A is stored to memory followed by virtual register B being stored to memory in a block then a connection between them is made equal to the execution cost of the block. If other spill instructions for A and B occur elsewhere in the control flow graph then the affinity between the two becomes stronger. Stack slots can be assigned to virtual registers by grouping those with the strongest connections. Complications include determining the strength of the connection between spill instructions in separate blocks. Also, sharing the same stack slot between virtual registers can affect relationships between spilled registers yet sharing slots is important for reducing cache misses.

An interesting point to be made is that the size of the cache line can play a role on the effects of spills. If a cache line is 64-bytes in size, then we may be able to spill 16×4-byte integers that would all fit into the same cache line. Given the increase in memory traffic the cache line is more likely to stay live in the cache. The performance penalty for such an allocation could be low given the fast speed of the cache.

Unfortunately, it can be difficult to guarantee that the current stack frame will be aligned on a cache line boundary. Arguments and callee saved registers will vary in the number that are saved to the frame. In addition, a single function may be called from several other functions, making the actual alignment with a cache line boundary

unpredictable. All functions in a program could force their stack frames to be aligned with cache line boundaries by computing an offset at compile time. However, this will only expand the amount of memory needed for the stack and potentially increase the number of cache line misses.

Chapter 3

Register Allocation Using Expected Distance

Register allocation is primarily focused on minimizing the effects of spills on running programs. This is approached by first trying to avoid spilling and, if that fails, by minimizing the costs of spills through the choice of what to spill and where to place the resulting spill instructions.

A spill decision is the problem of choosing which virtual register to spill when the number of live virtual registers exceeds the number of physical registers available. Heuristic based allocators, such as graph coloring or linear scan, consider spill choices incrementally, or one at a time. If a spill decision is required, a virtual register is chosen to be spilled based on some measure of spill cost with the goal of minimizing the overall costs. However, doing so incrementally does not guarantee minimal costs.

Virtual register live ranges overlap within functions. The choice to spill a particular live range will necessarily affect register pressure in other regions of the code and, potentially, affect other spill choices. Choosing to spill one virtual register over others may only be locally optimal due to the effects it can have on register pressure in other regions of the flow graph.

Heuristics for making spill decisions are often based on an estimate of the execution costs of the spill instructions required if that virtual register is chosen to be spilled. These estimates are generally inaccurate, or exaggerated, because the actual least cost placement of spill instructions cannot be known until allocation of registers is complete. Once again, the overlapping nature of virtual register live ranges along with register availability will affect spill instruction placement between spilled virtual

registers. Spill instructions could be placed precisely where the estimates predict, often at definitions and uses of virtual registers, but this is generally not the least cost placement.

Another approach to heuristic spill decisions is to consider the direction of program flow. It seems intuitive to suggest that virtual registers should be maintained in physical registers based on their order of execution. Given a choice between two virtual registers it is likely a better choice to spill the one that is used later than the other. There is a basis for this choice. The Bélády MIN algorithm, which chooses to spill the virtual register with the furthest next use, is known to be optimal for straight-line programs where all costs are equal. This has lead to its use in register allocation, although its application generally falls short of the original 'furthest next use' intent.

In this chapter we explore the use of Bélády's MIN algorithm as a spilling heuristic by computing the most accurate distance estimates yet. As originally defined for memory paging systems, page uses are considered sequentially. However, the control flow graph provides many possibilities for the actual next uses of a virtual register because branching is determined by input program data. We use branch probabilities to find the Expected Distance to next use which is the average distance over many program runs. We believe that this estimate is a marked improvement over past estimates that largely ignore path information when finding distances except where it is implicit in straight-line code.

We have designed a register allocator that solely uses Expected Distance for making spill decisions. We address several problems that affect the distance computations including differing spill costs between virtual register types, and determining distances when some edges have no next use. The MIN algorithm implies placement of reload instructions at the next use after being spilled but this is not necessarily the case for the control flow graph. We also present a novel method for optimizing spill instruction placement as well as a simple move coalescing method.

3.1 Bélády's MIN Algorithm

Bélády's optimal page replacement strategy for memory paging systems is also a popular heuristic in register allocation [12]. The MIN algorithm states that when memory is full and a page must be evicted to make room for another, then the optimal page to evict is the page with the "furthest next use." This will lead to an

optimally minimal number of page replacements provided the exact sequence of page uses is known prior to running the program. Unfortunately, it is not practical to run a program twice in order to achieve optimal performance on the second run.

In register allocation, the ordering of instructions within basic blocks, and their virtual register operands is known at compile time. Exact distances to virtual register next uses can be determined within basic blocks, although distances across block boundaries remain unknown. Complex control flow includes loops and conditional branching where execution paths are determined at run-time. The exact distance to next use at any given point cannot be guaranteed by the compiler. Therefore, it is not clear how distance to next use can be computed when the actual control flow path across conditional branches depends on data computed at run-time.

It must also be recognized that the optimality of the MIN algorithm also depends on the kinds of values that are evicted. The algorithm assumes that all pages have the same eviction costs. This is not the case in register allocation where virtual register spill costs can vary. Some virtual registers may have to be saved to memory — and are often called “dirty” — as their value must be saved to memory when spilled. Others are said to be “clean” as they can be recomputed without touching memory and are, therefore, less expensive to reload. These variances in costs for evicting, or spilling, a virtual register will certainly affect the costs of a spill and therefore the optimality of the furthest next use heuristic.

Given these complications, it is not clear that the MIN algorithm is useful for register allocation. Optimality can no longer be guaranteed. However, it can be approached and has been shown to be competitive, with better results than a graph coloring allocator on long basic blocks [40]. The challenge to using the MIN algorithm, as a viable heuristic for spilling, is in determining distances in the presence of complex control flow and varying spill costs.

A variant, the Furthest Nearest Next Use, has also been shown to have good results [20]. This heuristic, outlined in Figure 3.1, finds the nearest next use for each virtual register spill candidate and chooses to spill the furthest one. One advantage to this heuristic is that computing distances is fast because the next use resolves to a fixed distance regardless of control flow. However, the accuracy of the next use choice is likely to be compromised as the heuristic does not consider where control flow is likely to lead. A path with low or zero execution count, such as in error handling code, may provide the nearest next use. This should be a good spill candidate but could be made better by scaling the distance according to execution probability.

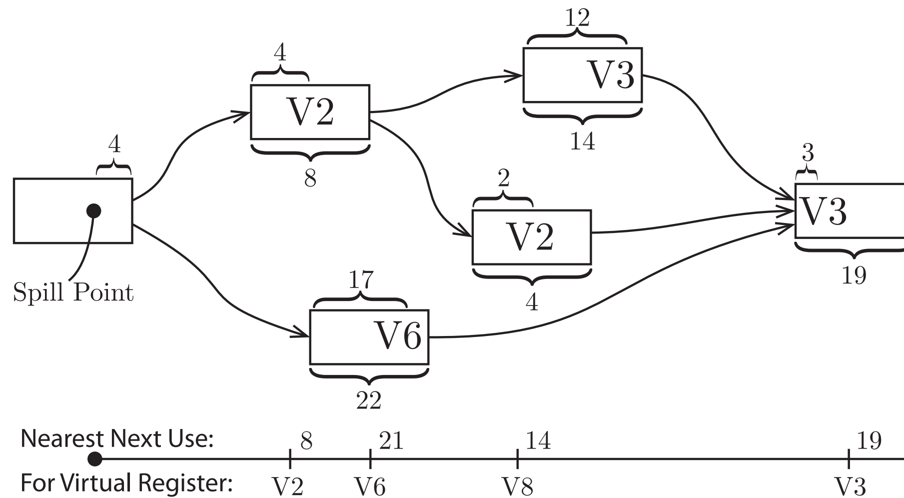


Figure 3.1: The Nearest Next Use heuristic would choose to spill V6 at the Spill Point. Block lengths are shown underneath each block and distance to virtual register uses within the block are shown on top. Computed distances to virtual register uses are shown on the bar at the bottom.

Example 3.1.1. Figure 3.1 shows a portion of a control flow graph with basic blocks and directed edges between them. Block lengths are shown underneath each block while the distance to the next use of the virtual register in each block is shown on top. The allocator has arrived at the spill point and is forced to spill a virtual register. There are four spill candidates: V2, V3, V6, V8. The distance to next use to each virtual register, based on the Nearest Next Use, determined by the allocator is shown on the line underneath.

There are several distances to virtual register V3 in the graph. The distance to V3 in the topmost block is 24 which is calculated from 4 (spill point to bottom of leftmost block) + 8 (through next block) + 12 (distance to V3 in topmost block). There are two paths V3 in the rightmost block. Passing through the block of length 22, the distance is $4 + 22 + 3 = 29$ while the other path is $4 + 8 + 4 + 3 = 19$. The Nearest Next Use to V3 is therefore $min(24, 29, 19) = 19$ which is taken to be the distance to next use for V3 by the allocator.

Given the distances indicated on the lower line, the allocator will choose to spill the virtual register with the furthest distance. For this example, V6 is chosen to be spilled.

It should be possible to more accurately estimate distances using branch frequencies. These frequencies can indicate which next use is more likely to be encountered

as execution moves forward. Unfortunately, new problems can arise. Zero frequencies and branches with no next use complicate the distance computations. Varying spill costs between virtual registers must also be handled. These problems must be solved in order to develop an effective heuristic.

3.2 Furthest Distance To Next Use

Finding exact distances cannot be guaranteed in the presence of conditional control flow. Different control flow paths may provide different distances. Instead, we can find the Expected Distance to Next Use which represents the average distance over a large number of runs of the program with typical input data. This distance can be computed as a sum of distances across outgoing branches from a block where the weights correspond to the probabilities of the different branches being taken.

The sum of weights approach has been suggested by Braun and Hack [20] but not described. They chose to use a Nearest Next Use heuristic across branches as an approximation of Bélády's MIN algorithm in their allocator. The nearest next use can be computed accurately and is guaranteed to converge to a fixed value. Computing a weighted average is more difficult because it requires an iterative approach that continually refines the value until a certain degree of accuracy is reached. When computing distances in the presence of loops the distance computed at the bottom of the loop across each successor edge will propagate to the top of the loop and back to the bottom, requiring another round of calculations. However, expected distance should lead to a better approximation as it considers the weights of all edges and effectively increases the the distance to next use on low probability paths. Virtual register uses with low probability are not over-represented as they can be with the Nearest Next Use heuristic. Braun and Hack do not use branch probabilities to compute distances with their nearest next use heuristic. They do, however, use a large distance multiplier on distances across loop exit edges. This effectively makes virtual register uses appear farther away outside of loops and better spill candidates than those that remain within the loop.

The accuracy of expected distance is dependent upon the quality of the branch probabilities used. Most compilers will estimate branch probabilities or edge weights as a guide for analysis. Well known heuristics exist for making these predictions [9, 10] that are quite good in practice. More accurate statistics could be gathered from branch execution counts obtained through runtime profiling typically found in Just-

In-Time compilers [6].

3.3 Distance Analysis

We define the Distance To Next Use of a virtual register as the expected number of instructions from a point in the control flow graph to the next use of that virtual register. If the next use is within the same basic block then the distance will be an exact count of instructions. Otherwise, the distance will be calculated as a weighted average of distances across all paths to all possible next uses.

This estimate assumes that each instruction takes the same amount of time to execute which is not always the case. Instructions that access memory will, almost always, take longer to complete than simple register-to-register moves. Another formulation of distance would, therefore, be to consider the number of clock cycles to next use. This would be more difficult to estimate given that load and store instructions, that touch memory, can vary in the time it takes to execute them on some architectures.

Distance Analysis is formulated as an iterative data flow analysis pass that computes the Expected Distance to Next Use for each virtual register in the flow graph at the top and bottom of each basic block. Distance is computed in two stages by first finding the Liveness Probability of a virtual register, which indicates the probability of encountering a next use, and then calculating the Expected Distance.

3.3.1 Liveness Probability

The Liveness Probability (LP) of a virtual register is the probability that a virtual register use will be encountered on some forward path from the current point in the flow graph. This value is interesting because it provides more information than the classical bit vector Liveness Analysis [46] which simply indicates whether there is a use on *any* forward path. For example, a virtual register may be a good spill candidate if it has a very low probability of being encountered. It would be more difficult to classify a virtual register as a good spill candidate if the liveness information simply indicates that there is a next use.

Liveness is based on branch probabilities and the existence of register uses and definitions in basic blocks. These facts are reflected in the data flow equations 3.1 and 3.2 for computing Liveness Probability. Equation 3.1 describes the liveness probability

$$LPTop_B[v] = \begin{cases} 0 & \text{if } B \text{ is the EXIT node} \\ 0 & \text{if first occurrence of } v \text{ in } B \text{ is a def} \\ 1 & \text{if first occurrence of } v \text{ in } B \text{ is a use} \\ LPBot_B[v] & \text{if } v \text{ does not occur in } B \end{cases} \tag{3.1}$$

$$LPBot_B[v] = \sum_{S \in succs(B)} (P_{B \to S} \times LPTop_S[v]) \tag{3.2}$$

where

$P_{B \to S}$	the probability that the branch from B to S is taken.
$succs(B)$	set of successor blocks of B.

Figure 3.2: Data Flow Equations for computing Liveness Probability.

for a virtual register v at the top of a basic block. If the next occurrence of v within the basic block is use of v then there is a 100% probability that it will be used. If the next occurrence is a definition then there is a 0% probability that it will be used. If there is no occurrence of v in the block then the probability of a next use anywhere within that block is the same as the probability at the bottom. Probabilities at the EXIT node will be zero since there are no virtual register uses after that point.

Equation 3.2 describes the liveness probability of register v at the bottom of a basic block. It is a sum of probabilities across each successor edge of the block. The probability across each successor edge is the probability that the branch is taken multiplied by the liveness probability at the start of the target successor block.

The equations can be computed iteratively as outlined in Appendix A. In the first iteration, Equation 3.1 is computed for each basic block in the flow graph in order to determine the liveness probability at the top of the block. Only the first two cases can be computed as the liveness at the bottom of the block is unknown. When implementing the pass, each virtual register is represented as a current estimate of the liveness probability and a flag that indicates whether the virtual register v is defined or used within the block. This flag is used to determine if the probability at the bottom of the block should be propagated to the top on subsequent iterations.

On subsequent iterations, liveness probability is computed using Equation 3.2, propagating the probabilities through those blocks where the virtual register is live but not referenced. The distance at the bottom of a block is a sum of the probabilities across each edge. The probability across an edge is equal to the probability at the

top of the successor block multiplied by the probability that the edge will be taken. The accuracy of the analysis will improve with the number of iterations. When loops are present several iterations may be required to reach a reasonable level of accuracy.

3.3.2 Expected Distance

The Expected Distance pass computes the distance to next use of a virtual register over all paths that contain a next use. It is defined to be infinity if no path contains a next use. The data flow equations are shown in Figure 3.3 and describe the expected distance at the top and bottom of a basic block.

$$EDTop_B[v] = \begin{cases} \infty & \text{infinity if v is not live at the entry to } B. \\ dist_v & \text{if first occurrence of } v \text{ in } B \text{ is a use.} \\ EDBot_B[v] + |B| & \text{if } v \text{ does not occur in } B. \end{cases} \tag{3.3}$$

$$EDBot_B[v] = \frac{\sum_{S \in succs(B)} (P_{B \to S} \times EDTop_S[v] \times LPTop_S[v])}{LPBot_B[v]} \tag{3.4}$$

where

$dist_v$	a count of instructions from the top of basic block B to the first occurrence of virtual register v in B.
$\|B\|$	the full distance through (or number of instructions in) basic block B.
$P_{B \to S}$	the probability that the branch from B to S is taken.
$succs(B)$	set of successor blocks of B.

Figure 3.3: Data flow equations for computing Expected Distance of virtual register v at both the top and bottom of basic block B.

The expected distance at the top of a basic block is described by Equation 3.3. If the first occurrence of a virtual register, in the block, is a definition then there is no next use of the register and the distance to next use is defined to be infinite. If there is a use of the virtual register in the block then the distance is a count of the number of instructions from the top to the instruction containing the first use. Finally, if there is neither a definition or a use of the virtual register within the block then the distance to next use is the sum of the number of instructions in the block and the expected distance at the bottom of the block. Just as with the liveness probability

```
1: B0:
2:     R8 = 12     ; D
3: B1:
4:     ...
5:     R8 = R8 - 1 ; UD
6:     CMP R8, 0   ; U
7:     BGT B1
8: B2:
9:     ... ; R8 is dead
```

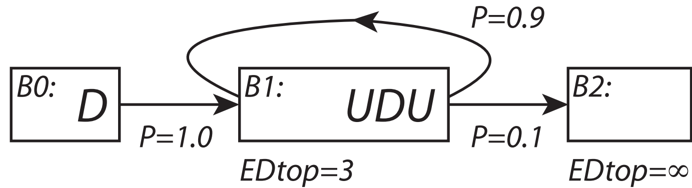


Figure 3.4: Example control flow graph showing a variable that is live from $B2 \rightarrow B1$ but is dead on $B2 \rightarrow B3$. D corresponds to a definition and U to a use of R8.

computation, it may be helpful to use a flag to indicate that no reference to the virtual register exists in the block.

Equation 3.4 describes the expected distance at the bottom of a block. It is computed as a sum of the distances across each successor edge of the block divided by the liveness probability at the bottom of the block. Dividing by the liveness probability at the bottom of the block gives the weighted average across those edges whose probability of being taken is non-zero. Across a successor edge, the distance is a product of the expected distance and liveness probability at the top of the successor block and the probability that the successor branch is taken.

3.3.3 Special Cases

Infinity While the use of infinity as a distance is descriptive, it is also problematic. If there is no next use across an edge, a distance to next use of infinity is assumed. However, when summing distances in Equation 3.4, if there is an infinite distance to next use across a non-zero probability edge then the resulting distance is infinite. Figure 3.4 illustrates the lifetime of a loop counting variable that exists within the loop. Since the probability of taking the loop back edge is very high, the expected distance to next use should be close to the distance of the first `U` in block B1. However, since the probability of the loop exit edge is non-zero, a simple averaging method would result in a distance of infinity.

$$\begin{aligned} EDBot_{B1} &= \frac{(P_{B1\rightarrow B1} \times EDTop_{B1}) + (P_{B1\rightarrow B2} \times EDTop_{B2})}{LPBot_{B1}} \\ &= \frac{(0.9 \times 3) + (0.1 \times \infty)}{0.9} \\ &= \infty \end{aligned}$$

In actuality, Equation 3.4 handles this case through the use of the Liveness Probability of the virtual register, albeit rather crudely. Since there is no next use across the loop exit edge, the liveness probability is zero. We require that $0 \times \infty$ be computed as 0 so that infinite distances are dropped from the calculation. As a result, for any non-zero probability across the loop back edge, the distance to next use will always be 3.

$$\begin{aligned} EDBot_{B1} &= \frac{\begin{array}{c}(P_{B1\rightarrow B1} \times EDTop_{B1} \times LPTop_{B1}) \\ + (P_{B1\rightarrow B2} \times EDTop_{B2} \times LPTop_{B2})\end{array}}{LPBot_{B1}} \\ &= \frac{(0.9 \times 3 \times 1) + (0.1 \times \infty \times 0)}{0.9} = \frac{2.7 + 0}{0.9} \\ &= 3 \end{aligned} \tag{3.5}$$

The problem of infinity bears some resemblance to the zero-frequency problem in data compression [13]. Adaptive encoding schemes assign probabilities to symbols based on how often they occur. Whenever a symbol is encoded the encoder must know what the probability is of all possible symbols at that point in the stream. If a symbol has not been encountered prior to the encoding point then there is no information with which to derive a probability. If zero is used for the probability then the encoding length, which is approximated as $-log(P)$, becomes infinite. Each new character must therefore be assigned a non-zero probability, but what should that probability be? In answer, Bell at al. [13, Chapter 3.1] state that

> It is possible to select robust methods that work reasonably well on most examples encountered in practice, but this is their only justification.

The best approximation is one that works "reasonably well." As written, Equation 3.4, is reasonable and robust in that it is not affected by infinite distances as paths that do not lead to next uses are ignored. It will always proved a finite distance if

there is at least one next use on some subsequent path. However, it is not satisfying because we would expect that if there are no next uses on some paths then the distance might increase.

It would be preferable if distances were scaled by the probability of a use being encountered. In Figure 3.4 the distance at the bottom of B1 should be close to $EDTop_{B1} = 3$ due to the high, 90% probability of the back edge. If that probability drops to 80% then the distance should be scaled to be farther, but not too much. However, if the probability of taking the back edge drops to 10% then the distance should be scaled more aggressively. We can achieve this behavior by dividing the distance result by the inverse of the sum of branch probabilities across dead edges. This is equivalent to dividing by the square of the liveness probability at the bottom of the block. Equation 3.6 computes the expected distance to next use at the bottom of block B1 using this estimator.

$$\begin{aligned} EDBot_{B1} &= \frac{\begin{array}{c}(P_{B1\to B1} \times EDTop_{B1} \times LPTop_{B1}) \\ + (P_{B1\to B2} \times EDTop_{B2} \times LPTop_{B2})\end{array}}{LPBot_{B1} \times LPBot_{B1}} \qquad (3.6) \\ &= \frac{(0.9 \times 3 \times 1) + (0.1 \times \infty \times 0)}{0.9 \times 0.9} = 3.33 \qquad (P_{B1\to B1} = 0.9, P_{B1\to B2} = 0.1) \\ &= \frac{(0.1 \times 3 \times 1) + (0.9 \times \infty \times 0)}{0.1 \times 0.1} = 30.0 \qquad (P_{B1\to B1} = 0.1, P_{B1\to B2} = 0.9) \end{aligned}$$

If the edge probabilities are reversed, such that the loop exit edge has a probability of 0.9, then the resulting distance is 30.0. This is much further away than 3.33 and indicates that the virtual register is a much better candidate for spilling than if we had not used the estimator and it had remained at 3.33.

Zero Probability Another situation that is similar to the problem with infinity, but with its own complications, is that of the zero probability edge. These edges may be discovered by a dynamic profiler when there are edges that are never crossed during execution and, subsequently, no execution count is recorded.

Figure 3.5 shows three examples where zero execution counts affect distances. If the Expected Distance equations are applied as they are defined in Equation 3.3 then Example *a* will result in a distance of 12 at the bottom of block B1 because the distance to B3 is zero and, therefore, ignored. Example *b* avoids the infinity problem and results in a distance of 3. Example *c* is more difficult to determine as

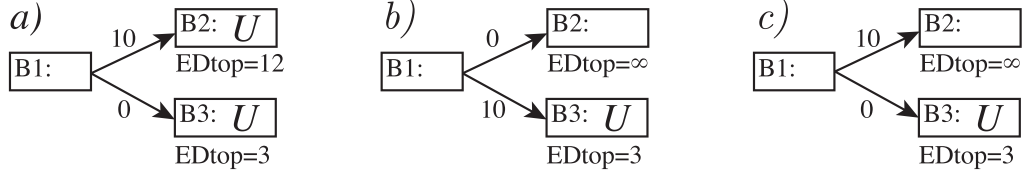


Figure 3.5: Three examples having a an edge with a zero execution count.

it would suggest that the distance to next use is infinity and therefore there is no next use at the bottom of block B1. However, when computing the expected distance for Example c using Equation 3.4 we get an error as shown in Equation 3.3.3. The Liveness Probability at the bottom of block B1 is $LPBot_{B1} = 0$ because there is no next use across the edge $B1 \rightarrow B2$ while the edge $B1 \rightarrow B3$ has never been taken. The probability of finding a next use is therefore computed as zero.

$$\begin{aligned} EDBot_{B1} &= \frac{\begin{array}{c}(P_{B1\rightarrow B2} \times EDTop_{B2} \times LPTop_{B2}) \\ + (P_{B1\rightarrow B3} \times EDTop_{B3} \times LPTop_{B3})\end{array}}{LPBot_{B1}} \\ &= \frac{(1 \times \infty \times 0) + (0 \times 3 \times 1)}{0} = \frac{0}{0} = error \end{aligned}$$

A next use across an edge with zero probability of being taken suggests that the next use will never be encountered because execution will never cross that edge. Its distance could be assumed to be infinite and, therefore, seen as having no next use. However, a zero probability indicates past experience, not future expectation. Using infinity, in this case, clouds its prior meaning as a distance to next use where there is none. It cannot be assumed that uses across zero probability edges will never be encountered without proving that the edges in question will never executed for any configuration of input data. Even when our profiler says the edge has never been executed we must assume that a future run may cross that edge.

To avoid a divide by zero error, one could simply not divide by zero. This would result in a distance of 0 at the bottom of block B1 which would suggest that the next use is at the bottom of the same block. As with the zero frequency problem in data compression there are multiple solutions. We offer one possible solution here.

A method for handling distances across zero probability edges is suggested by Bell et al. [13]. If there is at least one successor edge with a reported zero probability of being taken, then we add one to the execution count for each edge. For example, in

Figure 3.5 the edge probabilities become

$$P_{B1 \to B2} = 10/11 \approx 0.91 \quad \text{and} \quad P_{B1 \to B3} = 1/11 \approx 0.09$$

. Using these probabilities, the distance calculation is computed as below.

$$EDBot_{B1} = \frac{(0.91 \times \infty \times 0) + (0.09 \times 3 \times 1)}{0.09} = \frac{(0.09 \times 3 \times 1)}{0.09} = 3$$

This appears to exaggerate the distance to next use and has, in fact, been transformed into the infinity problem described earlier. In Figure 3.5, example b), the edge counts are reversed, there is a 100% probability of encountering a next use, and the distance to next use is the same as example a). However, in the case of Figure 3.5, example c), there is a use after both edges. The calculation is much more reasonable as below.

$$EDBot_{B1} = \frac{(0.91 \times 12 \times 1) + (0.09 \times 3 \times 1)}{1.0} = \frac{10.92 + 0.27}{1.0} = 11.19$$

Another possible case, not described in Figure 3.5, is that of the zero block execution count. All successor edges will have a zero count and thus a zero probability. In that case, it is straightforward and reasonable to assign an equal probability to each successor edge or $P = 1/|succs|$.

3.4 Allocation

Allocation of registers is performed as a local analysis on each basic block using the globally computed distance information. At the top of a block, the live virtual registers entering the block must be allocated to registers or memory. Allocation then proceeds from the top of the block to the bottom by considering the virtual register operands of each instruction. Register state information, or the location of each virtual register, is saved at both the top and bottom of each basic block.

Initial register placement at the top of a basic block and spill choices within are guided by the distance information. However, we do not use pure distance for these decisions but instead scale distances using multipliers based on the spill costs of each virtual register. Rematerializable virtual registers are better choices as spill candidates because they do not require store instructions. We can prefer them as spill candidates by making them appear more distant. Different kinds of rematerializable

values can be assigned different multipliers based on how expensive the corresponding spill instructions are to execute. Those that touch memory will have a lower distance multiplier. Rematerializing a move instruction that loads a constant value such as zero into a register is typically one of the least expensive operations on any given architecture and may be assigned the largest multiplier. Section 3.4.3 provides more explanation on this topic.

3.4.1 Input Register Allocation State

The first step in allocation of registers in a basic block is to compute the initial allocation on virtual registers that are live into the block. We call this a register state as it describes the current state of virtual registers in registers and memory. The initial allocation is computed from the output states of each predecessor block. Only those virtual registers that are live into a block will appear in the input state.

The main goal in computing the initial register state is to minimize execution costs. If a virtual register leaves a predecessor block in a register R then we try to place it in register R at the top of the successor block. If it arrives in a different register then a move instruction is required. The cost of this additional instruction is equal to the execution frequency of the edge between the predecessor and successor blocks. If the virtual register cannot be placed in any register then it transitions from R to memory at a greater cost due to the necessary insertion of a store instruction.

Algorithm 2 outlines the procedure for computing an input state to a basic block BB. We use the concept of a *Join* point where multiple outputs from predecessor blocks merge at the top, or input, to a basic block. This join contains a set of distinct input locations and the weights of the edges they arrive on. If a virtual register V arrives at a block in the same register location R on several edges then R is represented with the combined weight on all of those edges. If the weight is higher than all other distinct inputs for V then R may be the best candidate for placement. Algorithm 1 describes the collection of input join points for all virtual register inputs to a basic block BB.

In Algorithm 2 we are interested in placing virtual registers based on their distance. Inputs are considered in nearest distance first order. However, if an input is in memory on all incoming edges then it is placed in memory at the top of a block. There is likely no advantage to placing it in a register as the cost to place loads on each input edge is the same as placing a single load within the block. For those inputs that are

Algorithm 1 Algorithm for collecting the input virtual registers to a basic block. Each virtual register input is represented by a *Join* point that records the edge weight of each distinct input location. A location is either a particular register or memory.

```
 1: procedure CollectInputs(BB)
 2:     Joins ← empty list
 3:     foreach virtual register V in inputs(BB) do
 4:         J_V ← new Join for V
 5:         foreach basic block PBB in predecessors(BB) do
 6:             W ← edge weight of PBB → BB
 7:             L ← location of V at output(PBB)
 8:             if J_V contains location L then
 9:                 J_V[L] ← J_V[L] + W
10:             else
11:                 J_V[L] ← W
12:     return Joins
```

in a register on at least one input edge we consider placement in physical registers in order to minimize costs. For an input join J, our heuristic approach is to sort its input register locations by decreasing edge weight and attempt placement in physical registers in that order. If a candidate location for J already contains some join K then we check to see if K can be evicted based on the weight of its input edges for that location. If K is evicted then we try to place it into its next best candidate register. If a join has no more candidate registers then we choose any unallocated register. This method ensures that the virtual registers with the nearest next uses and arrive at the block input in at least one register are allocated first.

We consider register locations by the summed edge weights they arrive on in order to minimize register-to-register moves. This can be beneficial for minimizing their costs in the later move coalescing pass. However, there is some expense to collecting the inputs and sorting them by edge weight. It is possible to ignore register locations and simply allocate based on minimizing store and load instructions. In our experience, ignoring register locations does lead to more move instructions being inserted.

Example 3.4.1. Figure 3.6 shows an example of computing an input state from the output states of predecessor blocks. The locations of virtual registers must be matched at the input to BB5. Figure 3.7 shows the expected distances at the top of BB5 for the virtual registers that are live in. We consider them in the order `V0`, `V3`, `V2` and place `V0`, `V3` in registers, and `V2` in memory. Placement in particular

Algorithm 2 Algorithm for computing the input register state at the top of a basic block.

```
 1: procedure PlaceInputs(BB)
 2:     Joins ← CollectInputs(BB)
 3:     State ← new input register state for block BB
 4:     sort Joins by nearest distance first
 5:     foreach Join J in Joins do
 6:         if J is in memory on all inputs
 7:                 or State has no unallocated register for J then
 8:             place J in memory
 9:         else
10:             sort input register locations for J by decreasing weight
11:             index(J) ← 0
12:             while J is not placed do
13:                 if index(J) ≥ size(inputs(J)) then
14:                     R ← an unallocated register
15:                 else
16:                     R ← input(J, index(J))  ▷ get index(J) register choice for J
17:                 if register R is unallocated then
18:                     State[R] ← J
19:                 else
20:                     K ← State[R]  ▷ R is currently allocated to join K
21:                     if weight(K[R]) < weight(J[R]) then
22:                         State[R] ← J
23:                         J ← K  ▷ make while loop iterate again to place K
24:                     index(J) ← index(J) + 1
25:     return State
```

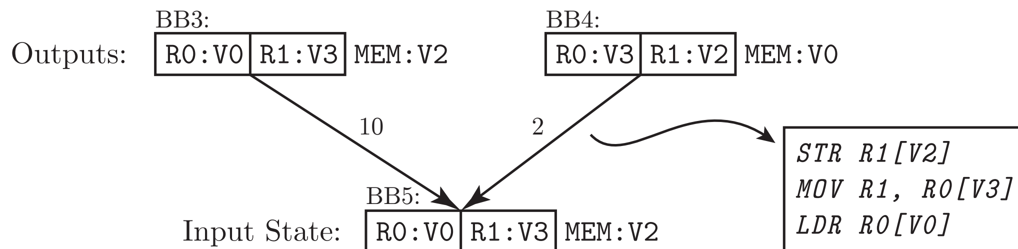


Figure 3.6: Computing the input state for basic block $BB5$ with two registers available. The output states of blocks BB3 and BB4 show the locations of virtual registers in both physical registers and memory. These are used to compute the input state to BB5. Spills are inserted on edge $BB4 \rightarrow BB5$ in order to match locations at the input to $BB5$.

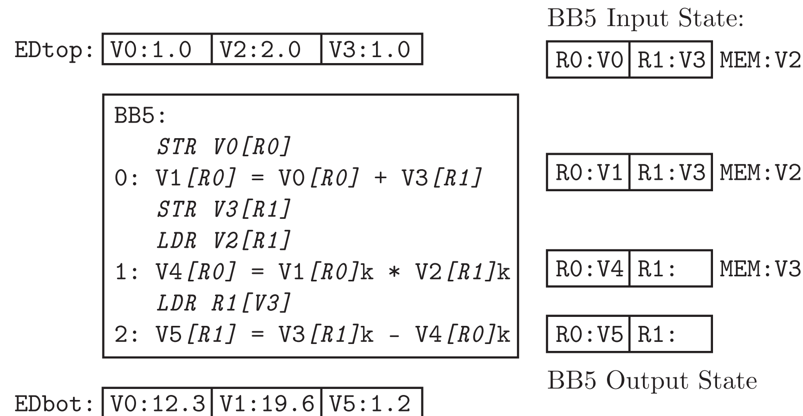


Figure 3.7: Allocation proceeds by modifying the register state and inserting spill instructions as necessary. Store and load instructions have been inserted into the block as required. The symbol `<k>` indicates a kill or last use of a virtual register.

registers is informed by edge counts. With edge $BB3 \rightarrow BB5$ being stronger, register placements follow from BB3. Since locations change on edge $BB4 \rightarrow BB5$ we must insert spill instructions in order to match locations at the top of BB5. `V2` is stored to memory, freeing register R1. `V3` is moved from register R0 to R1. `V0` is loaded into register R0. This ordering is important to prevent values being overwritten.

Once the input virtual register placements are decided, the "register state" is saved as the input state for the block. A copy of the input state also represents the current state which will be modified at each instruction to eventually form the block output state. By combining the input state with the output of a predecessor block, spill and move instructions can be determined along input edges.

3.4.2 Allocation Within Basic Blocks

Register state information is not saved within the block except in certain cases. Each virtual register use or definition in each instruction is assigned a physical register location. If a virtual register location is modified, such as a spill, reload, or move, then the corresponding instruction is inserted at that point in the block. The location of a virtual register at an arbitrary point within a block can be recomputed from the input or output register state information.

Allocation proceeds by visiting each instruction, from the top of the block to the bottom, and considering each virtual register operand. At each instruction, virtual register uses are considered first. The current register state holds the locations of the

live virtual registers. If the virtual register is defined in a physical register then that physical register is assigned to the virtual. If the virtual register location is in memory then a load instruction is issued immediately prior to the current instruction. Once the allocation of register uses is complete, any that are last uses of a virtual register, often labeled as a 'kill,' can be cleared from the current register state and their physical registers cleared. Register definitions are then assigned and their locations added to the current register state.

Algorithm 3 Method of computing the pure distance to next use of a virtual register V beginning at I_N.

```
 1: procedure ComputeDistance(block B, instruction I_N, virtual register V)
 2:     dist ← 1
 3:     foreach instruction I in B beginning with I_N do
 4:         foreach virtual register use U in I do
 5:             if number(V) = number(U) then
 6:                 return dist
 7:         foreach virtual register definition D in I do
 8:             if number(V) = number(D) then
 9:                 return ∞
10:         dist ← dist + 1
11:     return dist + distance to next use of V at bottom of B
```

At some point during allocation there may be no unallocated physical register available to hold a newly defined value. This forces a spill decision to be made. Some virtual register must be chosen to be evicted from a physical register. We choose to spill the virtual register with the furthest next use from the current point in the block. Since distance information has only been saved at the top and bottom of a basic block, we must compute distances within the block. Once again, we use the augmented distance that has been adjusted with distance multipliers based on spill costs, as described in Section 3.4.3.

Each virtual register in a physical register is considered as a candidate for eviction. The exact candidate is determined by computing the distance to next use for each candidate. Algorithm 3 describes the method for computing pure distance, without distance multipliers, to next use of a virtual register V within a basic block. We count the number of instructions in the block until we discover a next use or definition of the virtual register V. We begin at the next instruction I_N after the spill point which we assign a distance count of one. If we encounter a use of V prior to a definition then the distance is a count of the number of instructions to that point. If we encounter a

definition of V prior to a use of V then the distance is infinity because there is no next use. If neither a use or a definition of V is encountered then we reach the bottom of the block and return a distance that is the sum of the number of instructions counted and the distance at the bottom of the basic block. Distance multipliers can be added by determining the appropriate multiplier for the virtual register and multiplying the distance on return.

After deciding on which virtual register is to be spilled, we place a store instruction for the virtual register immediately prior to the current allocation point. Spill instructions within the block maintain the consistency of the code. However, the costs of these spills can, and will, be optimized during the Spill Analysis pass described in Section 3.5.

Example 3.4.2. Figure 3.7 shows an example of allocation of registers within a block. The register state is shown at each step after allocation has completed for the corresponding instruction. The input state is not modified during allocation. A separate state is modified at each step and saved as the output state for the block.

Line 0: Uses are allocated before definitions. `V0` and `V3` are assigned physical registers R0 and R1, respectively, from the current register state. Definitions are allocated next. A register is required for the definition of `V1` so either `V0` or `V3` must be spilled. Distances are computed by counting forward. `V3` is found on line 2 with a distance of 2. `V0` is not found in the block so its distance is computed as 2 + `EDbot[V0]` = 14.3. `V3` is the furthest next use and is spilled by inserting a store instruction prior to line 0. `V1` is defined into register R0.

Line 1: `V1` is assigned R0. `V2` must be loaded from memory. Since there is no available register one must be spilled. `V3` has the furthest next use so is chosen to be spilled. A store instruction is inserted. `V2` is loaded from memory into register R1. After the uses are allocated the virtual registers `V1` and `V2`, marked as last uses, or kills, can be cleared. Both registers are now empty. Virtual register `V4` is then defined into register R0.

Line 2: Virtual register `V3` is loaded from memory into register R1 by placing a load instruction prior to Line 2. Both virtual registers are last uses at Line 2 so both physical registers are cleared. `V5` is defined into register R0.

It can be helpful to use register preferences to guide the choice of registers and spills. If a virtual register's live range crosses a function call then the preferred physical register location might be a callee saved register. It will be preserved across

the call by the function being called if it is to be used. Conversely, if the live range does not cross a function call then the preferred physical register can be placed in a caller saved register which may be overwritten by the function being called. Specific instructions may impose their own physical register locations on a virtual register. By considering these requirements when defining registers then, at the very least, there can be a reduction in the number of move instructions.

Iterative Computation

A single pass over the blocks in a function may not be enough to accurately compute an allocation for a block. If a block is visited prior to any of its predecessors then there will most likely be insufficient information to compute an input state. This can be due to the order in which basic blocks are visited in the function or the presence of loops within the code. Several passes over the blocks in a function may be required to achieve the best accuracy. However, given that allocation on a basic block is not trivial, we must be careful to limit the revisiting of blocks to only those blocks where changes have been made.

We use the well defined Round Robin Worklist method [58, Chapter 8.4] to iteratively compute the allocation of basic blocks in a function. Iterative data flow analysis is briefly described in Appendix A. The worklist method uses a queue to record blocks that need to be revisited. An initial pass through the blocks in a function will add a block B to the queue where B has at least one predecessor that has not been visited. The block B has insufficient information to accurately compute its input state but is allocated anyway. When all blocks have been visited, the queue is serviced. A block B is removed from the queue and allocated. If its output register state has changed from the previous output state then all of its successor blocks are added to the queue so that changes can be propagated forward in the control flow graph. Note that a block is only added to the queue if it is not already in the queue.

We have stated that blocks are added to the queue if their output register state "changes" from the previous output state. It is important to consider what kind of changes are required as the strictness of the check can affect both the number of iterations required over the blocks but also the result of the allocation itself. The output state consists of a set of virtual registers where each is assigned a location in a specific physical register or in memory.

At its most strict, a change in output state is detected if a single virtual register

is in a different output location between the past and present states. This could be a transition between register R and memory M (store $R \rightarrow M$, load $M \rightarrow R$) or between different registers (move $R_X \rightarrow R_Y$). This check can result in a high number of iterations with no convergence since some register-to-register moves may be unavoidable. In some cases register numbers can oscillate between two virtual registers on successive iterations. It could be helpful to implement a mechanism for register preferences [21] whereby the allocator could be guided towards making certain register choices that minimizes the number of changes. However, even with preferences changes in output state may be unavoidable. Some stopping criteria is likely required such as stop iteration when a block is visited more than N times.

A less strict measure of change is to simply detect transitions between memory and register. This measure reduces the number of iterations required although it may result in more move instructions being inserted. This is the approach that we took with our implementation.

3.4.3 Spill Preferencing

The main problem with the MIN algorithm, when applied to register allocation, is that it assumes that all costs are equal. In register allocation they are not. Some virtual registers are rematerializable and should be preferred for spilling over those that require temporary storage to memory. Store costs can vary, for those that require memory storage, based on execution count and location. Store costs are not considered by the distance heuristic.

An effective means to incorporating the different costs of spilling between virtual registers is through the use of distance multipliers. Virtual registers can be categorized by the type of spilling required with each category assigned a different distance multiplier. Rematerializable virtual registers, with low spill costs, can be made to appear farther away using a large multiplier. This, in turn, can help make them appear as better spill candidates than those that require storage to memory. A load from a known memory address does not require a store so it would be preferable to spill these over a loads that do require stores. However, since they touch memory they would be assigned a smaller multiplier than a rematerializable virtual register.

Store costs are not easily applied as distance multipliers. Those virtual registers that require storing to memory are essentially the same — they must all be stored. Their difference comes from the potential execution costs of storing them to memory.

Once a virtual register is spilled, subsequent spills of that register may not be as expensive provided the new spill point is covered by the original spill. This poses a problem as there may be different spill costs, and therefore distance multipliers, over the range of the virtual register. A further complication is that optimization of spill placement is not performed until after allocation is complete. The placement of spill instructions assumed during allocation is likely quite different from what will be the optimized placement.

We were not able to find a reasonable method for incorporating store costs into the distance metric as multipliers. However, we do not entirely ignore store costs either. Distance multipliers already help to avoid issuing store instructions by increasing the distance to next use for those virtuals that do not require stores. In addition, we also optimize the placement of spill instructions in a post-allocation pass. This appears to be effective in reducing the costs of store instructions.

3.5 Spill Placement

The MIN algorithm was originally defined for a straight-line sequence of page replacements where page reloads occur immediately prior to their next uses and all costs, for both store and reload, are assumed to be equal. Register allocation on complex control flow differs substantially. Placement of spill instructions is less obvious because execution frequencies can vary between basic blocks. Costs can be optimized by moving store and reload instructions to less frequently executed blocks or branches.

During allocation, spill instructions are placed where required and are not moved to optimize costs. These can occur due to a spill decision within a basic block or on an edge in order to align the location of a virtual register into a basic block. Within a block, stores are placed where a virtual register is spilled to memory while reloads are placed immediately prior to the use that requires them. On edges, stores are inserted to ensure the virtual register is in memory at the top of a block while reloads ensure the virtual is in a common register. These placements ensure that virtual registers are consistent over their lifetimes and offer explicit information on where virtual registers transition between physical registers and memory.

There may be considerable range for spill instruction movement based on initial placements. Spill instructions could be moved anywhere that there is a physical register available to hold the virtual register value. However, we limit movement to anywhere that the virtual is in a physical register. For reload instructions this means

we can move them forward in the flow graph toward a use of the virtual register. Store instructions can be moved back in the flow graph towards definitions of the virtual register. We do not move these instructions in the opposite direction where the virtual register is in memory. This would mean reviving a virtual register into a physical register and would require additional analysis to determine which physical registers are available to place the virtual register into.

Store instructions can be moved backwards in the flow graph towards the definitions that cover them. However, the original store position must be covered by insertion of replacement store instructions on every path leading from a definition to that store position. Since loops generally have only one entry point, moving a store outside of a loop is likely to significantly reduce its dynamic execution costs. It should also be noted that a store instruction need not mark a transition of the live range from register to memory. A value can be "pre-stored" if it is effective to do so.

Load instructions are required by the uses that they precede and can be moved forwards in the flow graph towards them. However, in the case that a load immediately precedes a use there is no room for forward movement. We do not optimize the placement of these load instructions because we would need to revive the live range backwards in the flow graph at an additional analysis cost. However, this cost could be worthwhile in the case of reloads within loops. Moving them out of the loops could help to reduce costs. Load instructions, placed on edges, are subject to optimization because they do not immediately precede a use. These may be moved forward in the flow graph provided that each path from the original load to a use that it covers is intersected by a new load instruction.

Finally, rematerializable virtual registers do not require store placement so each store inserted during the allocation pass can be deleted. This may lead to orphaned definitions with no next use. These can be deleted as well. Optimizing the placement of rematerializable values is the same as load placement except that the instruction issued is the original defining instruction rather than a load.

3.5.1 Problem Description

We will call the problem of optimizing the placement of spill instructions a *Spill Problem*. The individual problems of optimizing Store and Reload instructions are constructed differently but are ultimately formulated in the same way. This allows us to solve both problems using the same algorithm. We will first describe the con-

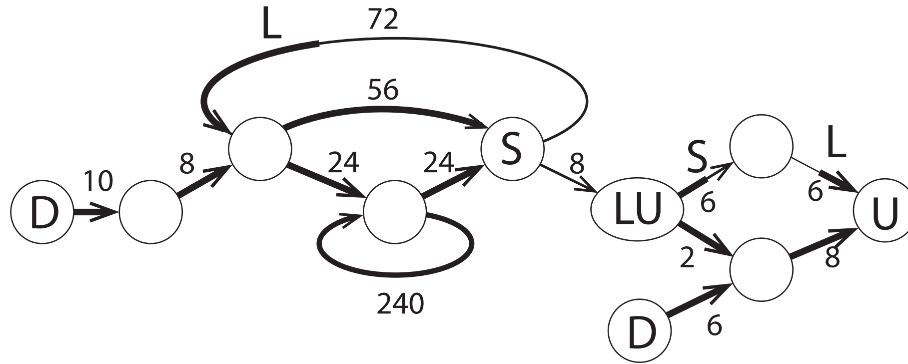


Figure 3.8: An example control flow graph after allocation and before spill placement optimization.

struction of the spill store problem with an example and follow it by a description of the load problem.

Figure 3.8 illustrates the live range sub-graph of the control flow graph where a spilled virtual register is live. This Control flows from left to right, except where indicated by a back edge, with the bold edges signifying that the virtual register is in a physical register along those paths. The graph contains register definitions (D) and uses (U) as well as spill stores (S) and loads (L) inserted by the allocator. Numbers represent the execution frequency of edges.

Definition 1 (Store Path)**.** A *Store Path*

$$P_S = (D, S, E_P)$$

is a straight-line directed sequence of edges $e \in E_P$ from a definition D of a virtual register to a spill store S for the same virtual register. The path is not intercepted by another spill store or a spill reload instruction and is, therefore, in a register on all edges $e \in E_P$ from D to S. The actual sub-graph is necessarily bounded by the edges and vertices contained in the live range for $VReg$.

Definition 2 (Store Graph)**.** A *Store Graph*

$$G_S = (D_S, S_S, V_S, E_S)$$

is a directed sub-graph of the live range of a virtual register $VReg$ on the control flow graph. Every edge $e = (x, y) \in E_S$ is a member of a store path P_S from a definition of $VReg$ to a spill store instruction for $VReg$. For an edge $e = (x, y) \in E_S$, the

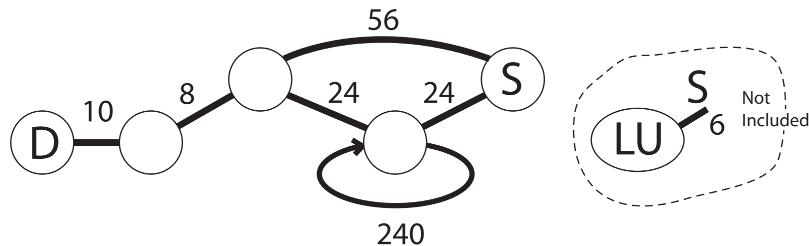


Figure 3.9: The Store Graph for the control flow graph of Figure 3.8.

vertex endpoints x and y are both in the set of vertices V_S. D_S is the set of virtual register definitions in the sub-graph located in vertices. S_S is the set of spill store instructions in the sub-graph which may either represent a vertex (store instruction in a basic block) or an edge (store instruction on a branch between blocks).

We use the term *Store Graph* to describe the sub-graph of the live range of a spilled virtual register where store instructions can be moved. The store graph for Figure 3.8 is shown in Figure 3.9. This graph has a number of *Store Paths.* As an example, the path $P = (D, S, 10, 8, 56)$ described by the weighted edges in Figure 3.9 form one such path from definition D to store S. This path is, initially, intercepted by S at its endpoint. If the placement of store instructions for the entire store graph is optimized, with the likely deletion of S, then every path from D to S, including P, must be intercepted by a spill store instruction. Note that there is no path from D to the second store in the example graph for the simple reason that all paths leading to that store are intercepted by the first store instruction.

The *Store Problem* is the problem of finding an optimal placement of spill store instructions on the store graph in order to minimize execution costs. This problem is naturally represented as a network flow problem where the edges and vertices in the store graph form the network on which we wish to find the placement. To formulate the store problem as a network flow problem we first define the *Store Problem Graph.*

Definition 3 (Store Problem Graph)**.** A *Store Problem Graph*

$$G_{SP} = (Source, Sink, (D_S, S_S, V_S, E_S))$$

is a *Store Graph* augmented with source *Source* and sink *Sink* nodes. *Source* is con-

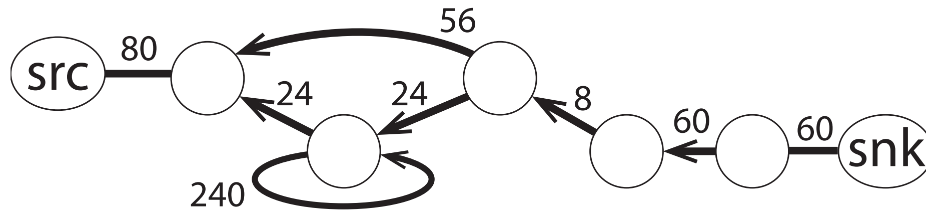


Figure 3.10: Store problem graph showing the minimum cut (dotted line) solution.

nected to each spill store instruction $s \in S_S$ and *Sink* is connected to each definition $d \in D_S$.

Figure 3.10 shows a store problem graph for the corresponding graph in Figure 3.9. This new graph contains a source and a sink node that indicate direction of flow, and therefore spill instruction movement, from spill store instructions to definitions. Note that this direction of movement is in the opposite direction to that of the original control flow graph.

A *Load Problem* is almost identical to the Store Problem. Spill reload instructions are moved from source to sink on the Load Problem Graph. This corresponds to moving the spill reload instruction from their original location to the next uses in the graph. Unlike the Store Graph, the direction of edges in the Load Problem Graph is identical to the original control flow graph reflecting their forward movement toward uses of the virtual register.

Definition 4 (Load Path)**.** A *Load Path* $P_L = (L, U, E_P)$ is a straight-line directed sequence of edges $e \in E_P$ from a spill reload instruction L of a virtual register to a use U of the same virtual register. The path P_L is not intercepted by another spill store, spill reload, or use of the virtual register. The virtual register is, therefore, live in a physical register from L to U on all edges $e \in E_P$.

Definition 5 (Load Graph)**.** A *Load Graph*

$$G_L = (L_S, U_S, V_S, E_S)$$

is a directed sub-graph of the live range of a virtual register $VReg$ on the control flow graph. Every edge $e = (x, y) \in E_S$ is a member of a load path P_L from a spill reload of $VReg$ to a virtual register use of $VReg$. For an edge $e = (x, y) \in E_S$, the vertex endpoints x and y are both in the set of vertices V_S. L_S is the set of spill reloads in the sub-graph that are located on edges in E_S. U_S is the set of virtual register uses in the sub-graph which are located in vertices (in a basic block).

Only edge loads are specified in our definition of the load graph. This is in keeping with our earlier decision to leave block loads immediately prior to the next uses. Since we only allow load instruction to move forward in the graph, block loads cannot be moved.

Definition 6 (Load Problem Graph)**.** A *Load Problem Graph*

$$G_{LP} = (Source, Sink, (L_S, U_S, V_S, E_S))$$

is a *Load Graph* augmented with source *Source* and sink *Sink* nodes. *Source* is connected to each spill reload instruction $l \in L_S$ and *Sink* is connected to each use $u \in U_S$.

3.5.2 Solving

Every path from source to sink in a Spill Problem must be intercepted by a spill instruction whose placement is being optimized — a store for a store problem and a load for a load problem. Therefore, a placement of spill instructions on the problem graph will necessarily represent a cut of that graph.

Definition 7 (Cut)**.** A *Cut* $C = (S, T)$ of a graph $G = (V, E)$ with weighted edges E is a partitioning of the vertices V into two non-empty sets S and T. The *Cut Set* of C_S is a set of edges where each edge $e \in C_S$ has one vertex endpoint in S and the other endpoint in T.

The weight of the cut is equal to the sum of the weights of all edges in the cut set C_S. The weights correspond directly to execution frequency estimates on each edge and form an estimate of the execution cost of the spill instruction placement.

There are likely to be many different cuts of a problem graph. Since a cut represents a placement of spill instructions, and we are interested reducing placement costs, our primary goal is in finding a cut with the minimum cost over all possible cuts. This is called a *Minimum Cut* of a graph G. The dotted line in Figure 3.10 represents a minimum cut of that graph.

Finding the minimum cut on a region of the control flow graph is an effective way to finding minimum cost placements of instructions. A separate compiler optimization pass known as Partial Redundancy Elimination, the compiler seeks to reduce the number of repeated computations in the code by removing duplicates and moving

them to less frequently executed regions of the graph. Cai and Xue [26] and Xue and Cai [75] use edge profiling information and the minimum cut to find the optimal placements for instructions such that the number of times a computation is executed is minimized. In register allocation, Ebner et al. [35] use the minimum cut to minimize the number of load instructions, due to spills, that are executed. They combine all of the load instructions placement problems for each spilled virtual register in a function into a single graph. Placements are then solved optimally using integer linear programming. While they consider store costs in the solver they do not solve the placement of store instructions.

Our approach seeks to optimize the placement of both store and load instructions in order to minimize execution costs. We solve each store and load placement problem independently of each other and of other spilled virtual registers. A consequence of this approach is a loss of optimality since solving placement for one virtual register can affect that of others, especially when physical registers become available due to movement of a spill instruction. We must also consider the relationship between store and reload placement for each virtual register. Moving load instructions forward in the flow graph can lead to a larger region that must be covered by store instructions, may include other definitions not included in the original problem, and potentially increase store costs. We do not consider these possibilities, opting instead to focus on minimizing reload costs. Some parts of a live range may occur later during the move coalescing stage.

Many methods exist for solving a minimum cut problem on a graph [45, 70, 30]. When a graph has several minimum cuts, all having the same cost, these methods will find one of them. For our spill problem, cost is not the only factor in making a minimum cut a best cut. Where execution costs are identical between cuts, we prefer the cut that expands the area that the virtual register is in memory. This preference can help to reduce register pressure across the entire control flow graph and, while we don't try to prevent spills at this stage, it can help to reduce move instructions and may allow other virtual registers to be revived during the later move coalescing stage.

The solution of the minimum cut on the graph will be a set of edges. This implies that spill instructions will be placed on edges and not within existing basic blocks. The observation that drives this decision is the fact that the total cost of a minimum cut that only includes edges is less than or equal to a minimum cut that contains basic blocks. The proof of this statement relies on the fact that a weighted edge

leading into or out of a basic block will have a weight that is less than or equal to that of the basic block. The basic block may have more than one edge entering or leaving and will therefore have a weight that is greater than or equal to that of a single edge. Therefore a minimum cut should prefer to cut edges rather than basic blocks. However, spills will only be placed on critical edges, or edges whose predecessor block has multiple outputs and successor block has multiple inputs. If the edge is not critical then the spills can be moved into the predecessor or successor blocks. Even if the edge is critical, the spill can be moved into another block if it can be merged with spills on all other edges.

Contraction

Our Minimum Cut solver relies on Graph Contraction as a means to reduce the size of the graph. Contraction involves the controlled removal of edges from the graph where it has been determined that they will not be part of the solution. An edge e with endpoints u and v may be removed from the graph and replaced by the vertex w. All other edges incident to u and v are added to w. Once all vertices are removed from the graph only the solution edges, connecting source to sink, remain.

Graph contraction is used in graph coloring register allocators to coalesce virtual register nodes in the graph in order to eliminate move instructions [22]. Contraction has also been used to find the minimum cut of a graph. Notably, Karger [44] describe a randomized algorithm for finding a minimum cut. The algorithm was later refined by Karger and Stein [45]. The basic idea is to contract edges in random order to find a cut of the graph. If run a sufficient number of times there is a high probability that one of the cuts found will be a minimum cut of the graph.

Our contraction method does not use randomization in the general case, preferring to apply specific rules to patterns in the graph. By carefully designing the rule set we are able to preserve the optimality of the solution. Rules based approaches to contraction have been used before. Guattery and Miller apply this approach to planar directed graphs where nodes can be arrange on the plane such that no two edges cross each other [39].

Figure 3.11 shows three example contraction rules. Each example shows a portion of a control flow graph, with basic blocks and directed edges, and its contracted form.

In example a), the edge with weight 7 is removed from the graph because it leads to a leaf node rather than a source or sink. Example b) shows the removal of the edge

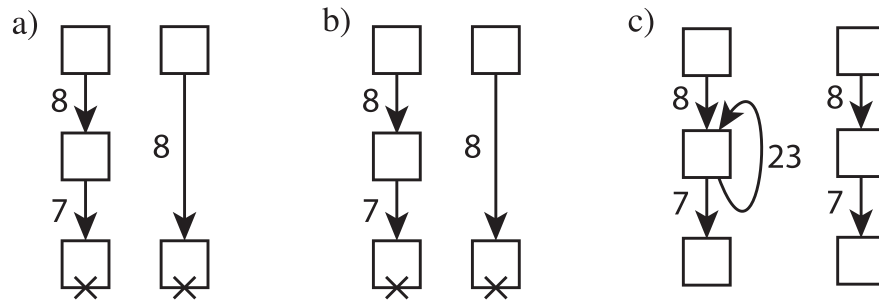


Figure 3.11: Simple edge reductions showing the flow graph, and its reduction: a) leaf nodes, b) sequential edges, c) back edges.

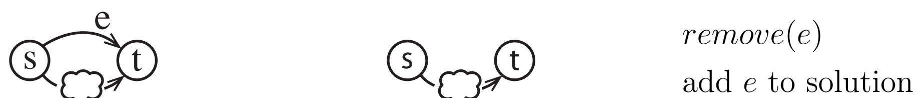
with weight 8 because a minimum cut will prefer the edge with the lesser execution frequency. In example c), the loop edge can be removed because a minimum cut will not intersect it, preferring either the incoming or outgoing edge to the loop instead.

Contraction Rules

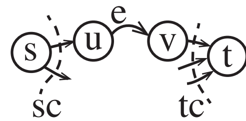
We use a small set of contraction rules that are applicable to arbitrary control flow graphs. Applying these rules alone, will maintain the optimality of the resulting minimum cut of the graph and will not degrade the cost of the solution. However, these rules alone are not sufficient to guarantee a solution to an arbitrary graph. We will address this problem after describing these rules.

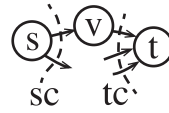
Each rule is defined with respect to an edge e having endpoints u and v. On each contraction step an edge is removed from the graph. In some cases the edge e is simply deleted from the graph while in others its endpoints u and v are merged into a single vertex. Deleting an edge means removing it from the graph and from its endpoints u and v. Merging vertices involves transferring all incident edges from one vertex to the other. Note that the source and sink nodes are special cases. A vertex can be merged into the source or sink but not the reverse. The source and sink cannot be merged together. Instead, an edge connecting them is removed from the graph and is added to the solution.

a) **Source to Sink** — If there is an edge e from source s to sink t then that edge must be part of the solution since a cut of the graph will intersect it. The edge is removed from the graph and added to the solution.



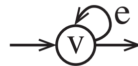
b) **Edge Weights** — If there is an edge e whose weight is greater than the current minimum cut W_{cur} then that edge can be removed from the graph as it will not be part of the solution. The current minimum cut is the minimum of the weight of the source cut W_{sc} (all edges leaving the source node) and the weight of the sink cut W_{tc} (all edges entering the sink node).

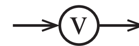




$W_e > min(W_{sc}, W_{tc})$
$remove(e)$
$merge(u, v)$

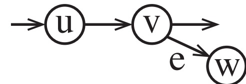
c) **Loops** — If there is an edge whose successor node is the same as the predecessor node then the edge can be deleted as the minimum cut will not intersect the edge.

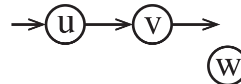




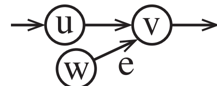
$remove(e)$

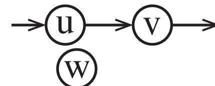
d) **Leaf Node** — If there is an edge whose successor node has no outgoing edges and is not the sink vertex then the edge may be deleted as well as the leaf vertex. A cut will not include edges to a leaf vertex as it could easily go around the vertex.





$remove(e)$



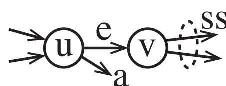


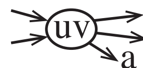
$remove(e)$

e) **Vertex Predecessor Merge** — If a vertex V has a single predecessor edge e where

$$W_e > \sum_{S \in succs(V)} W_S$$

weight is greater than or equal to the weight of all successor edges of V then vertices U and V along edge e can be merged. A minimum cut will prefer to cut the successor edges of V rather than the edge e.



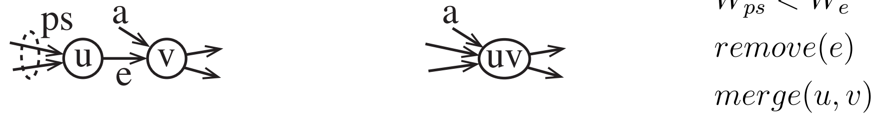


$W_e \geq W_{ss}$
$remove(e)$
$merge(u, v)$

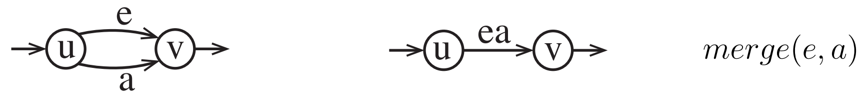
f) **Vertex Successor Merge** — If a vertex V has a single successor edge e and

$$W_e > \sum_{P \in preds(V)} W_P$$

the weight of the edge is greater than the weight of all predecessor edges into V then vertices U and V along edge e can be merged. A minimum cut will prefer to cut the predecessor edges of U rather than the edge e.



g) **Duplicate Edges** — If there are two edges with the same endpoints and destination then those edges can be merged into a single edge. If a cut of the graph intersects one of the edges then it must intersect both. The new edge represents the locations of the original edges and has a weight that is equal to the sum of the weights of the original edges.



h) **Cycle Edges** — If there is an edge a with endpoints u and v and an edge b with endpoints v and u then we say that these edges form a cycle. These edges are representative of loops in the original code with one of them likely being a back edge. It does not make sense for a minimum cut to intersect both as one edge covers the other. If it could be determined if one edge is a back edge then that edge could be deleted. Instead we merge along one of the edges. The other edge becomes a loop edge and may be removed by rule c.

Contraction can fail to reduce the graph because none of the rules described above is applicable. Those rules are simple with respect to a single edge. More complex rules may be possible that maintain the optimality of the solution but require analyzing a greater region of the graph. We have not investigated further, preferring instead to keep analysis simple and very localized in order to minimize the processing power required.

If contraction stops before a solution can be found then it is possible to determine an optimal minimum cut solution using a known minimum cut algorithm. These algorithms generally work in polynomial time and will certainly benefit from the reduced problem.

We avoid the complexity of implementing another minimum cut method, and possibly lose the promise of optimality, by using a simple heuristic. When contraction fails we contract an edge with highest cost (there may be several) and then try to apply the previous rules. The idea here is that by removing an edge with high cost, it will not add to the solution. However, we cannot guarantee that this will not increase the costs over an optimal minimum cut.

Algorithm

We use a simple algorithm to check for and perform the contractions of Section 3.5.2 as outlined in Algorithm 7. On each step of the algorithm at least one edge is removed from the graph. This continues until either the source or sink node has no incident edges.

The graph consists of vertices and edges. An edge is directional from vertex U to V with a $Weight$ and a set of $Locations$ in the original flow graph.

$$Edge =< U, V, Weight, Locations >$$

Edges can be merged during contraction and thus can represent multiple locations. The weight of an edge is equivalent to the execution frequencies of the locations the edge represents. A vertex has input and output weights, and input and output edge counts.

$$Vertex =< Weight_{in}, Weight_{out}, Edges_{in}, Edges_{out} >$$

The source and sink are special vertices where the source has no input edges while the sink has no output edges.

When computing the minimum cut, the graph is represented by a list of edges. Using a link list allows for the removal of edges in a single operation. Vertices are not directly considered and only exist in the graph if they are an endpoint of a edge in the list.

Operations on the graph are limited to those required by the rules in Section 3.5.2. Vertex related operations include checking for the number of input or output edges

and checking the input or output edge weights. These values are maintained in each vertex and will be updated whenever an edge having the vertex as an endpoint is modified. Vertices may also be merged into a single vertex. As a result the edges of one vertex must be transferred to the other. This can be handled by iterating over the list of edges and updating each edge where one of its vertices is an endpoint as outline in Algorithm 6. This is a $O(n)$ operation on the number of edges. The cost could be reduced by maintaining list of edges adjacent to each vertex so that only those edges would need to be updated.

Edge operations including checking weight, merging two edges into one, and removing an edge from the graph. Removing an edge from the graph will mean updating its vertex endpoints and deleting the edge from the edge list. This is outlined in Algorithm 5. Merging two edges simply means removing one edge and adding its locations and weight to the other edge. The weights and counts of the vertex endpoints must be updated as well.

Checking for duplicate edges (rules g and h) in Section 3.5.2) is a $O(n^2)$ operation as each edge must be checked against the others. This cost could be dramatically reduced by using an adjacency matrix [58]. The matrix is a two-dimensional square array indexed by the predecessor and successor vertex numbers of the edge endpoints. Vertices must be assigned a unique number in the range $0...n-1$ for n vertices. For our purposes, each element in the array is a reference to the actual edge corresponding to the vertex endpoint indices, or a null indicating no edge. This means that there is only one representation for an edge. Adding a duplicate edge to the matrix would be immediately obvious. Contraction rule g) in Section 3.5.2, for duplicate edges in the same direction, need not be checked as it would be seamlessly handled during a merge of two vertices.

Algorithm 7 will find a cut of the input spill problem graph and return a set of edges corresponding to placements in the original flow graph. The algorithm continues to contract the graph, according to Line 3, until either the source or the sink becomes detached with no incident edges. Edges connecting the graph to the detached source or sink were removed using contraction rule a) in Section 3.5.2.

Line 4 chooses a rule from Section 3.5.2 that is applicable to the edge E. In general, the checks are quite simple for rules a) through f). The expense of rules g) and h) depends on the method for duplicate checking which is largely eliminated when using an adjacency matrix described above. A straightforward method of checking edges is to sequentially iterate over the edges in the edge list. This has a worst case

Algorithm 4 Remove an edge from the graph

```
 1: procedure AddEdgeToOutputOf(Graph,E,V)
 2:     let U ← predecessor(E)
 3:     outweight(V) ← outweight(V) + weight(E)
 4:     outcount(V) ← outcount(V) + 1
 5: procedure AddEdgeToInputOf(Graph,E,V)
 6:     let U ← predecessor(E)
 7:     inweight(V) ← inweight(V) + weight(E)
 8:     incount(V) ← incount(V) + 1
 9: procedure RemoveEdgeFromOutputOf(Graph,E,V)
10:     outweight(V) ← outweight(V) − weight(E)
11:     outcount(V) ← outcount(V) − 1
12: procedure RemoveEdgeFromInputOf(Graph,E,V)
13:     inweight(V) ← inweight(V) − weight(E)
14:     incount(V) ← incount(V) − 1
15: procedure RemoveEdge(Graph,edge E)
16:     let (U, V) ← (predecessor(E), successor(E))
17:     call RemoveEdgeFromOutputOf(Graph,E,U)
18:     call RemoveEdgeFromInputOf(Graph,E,V)
19:     delete(E) from Graph edge list
```

Algorithm 5 Merging edge E into F

```
1: procedure MergeEdges(Graph,edge E, edge F)
2:     let (X, Y) ← (predecessor(F), successor(F))
3:     call AddEdgeToOutputOf(Graph,E,X)
4:     call AddEdgeToInputOf(Graph,E,Y)
5:     locations(F) ← locations(F) + locations(E)
6:     weight(F) ← weight(F) + weight(E)
7:     call RemoveEdge(Graph,E)
```

Algorithm 6 Merging two vertices

```
 1: procedure MergeVertices(Graph,U,V)
 2:     foreach edge E in Graph edge list do
 3:         if E has an endpoint U then
 4:             if U = predecessor(E) (ie E = (U,W)) then
 5:                 call RemoveEdgeFromOutputOf(Graph,E,U)
 6:             if U = successor(E) (ie E = (W,U)) then
 7:                 call RemoveEdgeFromInputOf(Graph,E,U)
 8:             let F ← edge E with endpoints U replaced by V
 9:             if edge F already exists in Graph then
10:                 Add locations(E) and weight(E) into edge F
11:             else
12:                 if U = predecessor(E) (ie E = (U,W)) then
13:                     call AddEdgeToOutputOf(Graph,E,V)
14:                 if U = successor(E) (ie E = (W,U)) then
15:                     call AddEdgeToInputOf(Graph,E,V)
16:                 change endpoints U to V in edge E
```

Algorithm 7 Contract Edges in a Spill Problem Graph

```
1: procedure GraphContraction(Graph,Edges)
2:     let Solution ← ∅
3:     while Source and Sink vertices of Graph have incident edges do
4:         if an edge E can be contracted using a rule in Section 3.5.2 then
5:             Contract edge E by performing associated action from
               Section 3.5.2 (remove, merge, add to Solution)
6:         else
7:             Choose an edge E with highest weight
8:             Merge the endpoint vertices of edge E
9:     return Solution
```

of $O(n^2)$ although in practice it is far less.

If no edge can be contracted using one of the contraction rules then Line 4 chooses an edge with highest weight. There may be more than one. A good choice would be one that is connected to either the source or sink, if possible, as that may reduce the cost of the current minimum cut on the remaining graph. This could allow rule b), where an edge weight is greater than the current minimum cut, to be applied on other edges.

On Line 9 the solution, consisting of a set of locations for the placement of spill instructions, is returned. These locations were added to the solution on Line 5 when rule a) was applied.

The locations in the solution can be on edges or within blocks. However, we wish to avoid placing spills on edges because this often means adding a new basic block to the flow graph where it may not be necessary. If the predecessor block has one outgoing edge then the spill may be moved to the bottom of the predecessor. Similarly, if the successor block has one input edge then the spill may be moved to the top of the successor. There are no additional costs to moving the spill in those cases.

Critical edges pose a problem because the spill can not be moved to either the predecessor or successor without increasing costs. A store could be moved to the predecessor but may not be movable to the successor if the virtual register is in memory on other input edges. A load could be moved forward provided the virtual register has been stored prior to the successor block on all input edges. A load may not be moved back if the specified register is not available at the bottom of the successor block. We simply split the critical edge and insert a new basic block with the added spill instructions.

Further optimization or cleanup is possible. Some live ranges may have multiple definitions. While the entire live range may not be considered rematerializable, some of its definitions may be. If a reload instruction is covered by a single rematerializable instruction then it may be converted into that instruction and a store and reload from memory can be avoided.

Our algorithm relies on profiling information almost exclusively to optimize placement. The only exception is the preference for edges that are closer to the sink than the source. This means that the algorithm is dependent on the quality of profiling information provided. Static information such as loop depth could be exploited to improve solutions. If each node is marked with a depth counter then entire loops

could be eliminated from the graph very quickly. The assumption here is that edge weights will be higher inside of loops and will be proportional to the loop depth. However, this is not necessarily true. Loop depth is a feature of the graph and may not reflect profiling information. It is entirely possible that code within a loop is rarely executed. We found that eliminating edges based on a rule similar to rule b) except using loop depth did not help to minimize costs.

3.6 Move Coalescing

The Move Analysis pass optimizes the number and placement of register-to-register move instructions. Move instructions are one of the least expensive to execute yet an excess of moves can lead to code expansion, wasted clock cycles, and a potentially noticeable effect on execution speed. Code expansion can increase the probability of cache misses which further reduce performance.

Moves may be inserted by an allocator for a number of reasons. Some of these are outlined in Figure 3.12. When multiple paths merge at the top of a block, all inputs must be in the same register. Move instructions may have to be inserted on incoming edges to ensure a virtual register is in the same physical register across all paths entering a block. In the example, virtual register V0 is defined in R0 in BB#4. It must be moved into R1 prior to branching to BB#6 in order to be in the same place as at the bottom of BB#5.

Move instructions can also be inserted within basic blocks in order to force a value out of or into a particular physical register. Function calls can create a number of problems due to calling conventions. In the example, virtual registers V0 and V1 must be preserved across the function call. Assuming that registers R0 to R3 are caller saved registers that are also used as function call argument registers, V0 and V1 are moved into registers R8 and R7 respectively, that will be preserved across the call. V1 is also an argument to the function call so it is copied into the first argument register R0. Most of these moves can be eliminated with a better register assignment. Only a single move is required where the live range of V1 splits around the function call.

The move coalescing problem is naturally described by an interference graph [29, 38]. This approach is effective at modeling the interferences between virtual register live ranges, or which ones are live at the same time and where, and of the connections representing move instructions between portions of live ranges. The move coalescing

```
 1: BB#4:                   BB#4:
 2:    V0 = ...                R0 = ...
 3:    JMP BB#6                JMP BB#35
 4:                         BB#35:
 5:                            R1 = MOV R0 ; align with BB#5
 6:                            JMP BB#6
 7: BB#5:                   BB#5:
 8:    V0 = ...                R1 = ...
 9:    JMP BB#6                JMP BB#6
10: BB#6:                   BB#6:
11:    V1 = V0 + 12            R2 = R1 + 12
12:                            R8 = MOV R1  ; preserve V0
13:                            R7 = MOV R2  ; preserve V1
14:                            R0 = MOV R7  ; copy to argument
15:    V3 = Func( V1 )         R0 = Func( R0 )
16:    V4 = V3 + V1            R1 = R0 + R7
17:    V5 = V3 + V0            R2 = R0 + R8
18:    ...
```

Figure 3.12: Showing an allocation of physical registers, including move instructions, for the input code with virtual registers.

problem using an interference graph is not the same as the graph coloring register allocation problem since an assignment of physical registers to virtual registers is already known. Spilling is not required.

The move interference graph is built by analyzing the ranges over which virtual registers are live in a register. Figure 3.13 shows an interference graph for the corresponding intermediate code. The virtual register V0 is broken out into six nodes over its lifetime in the graph. These nodes indicate the range over which the virtual register is in a single physical register within a basic block. A single node could contain a definition and multiple uses of a virtual register. Some nodes may represent a fixed physical register. They are the same as other nodes except they cannot be assigned a different physical register. When constructing the graph all instances of the same fixed physical register can be merged into a single node.

A move-related edge, connecting two nodes in the graph, indicates where a move instruction could be placed in the control flow graph. As such, they are assigned an associated execution cost based on where they would be placed. Each node for V0 in Figure 3.13 is connected by a move-related edge. Move-related edges are also added where move or copy instructions exist in the intermediate code. These edges can connect different virtual registers.

Interference edges connect nodes that are live at the same time and cannot be

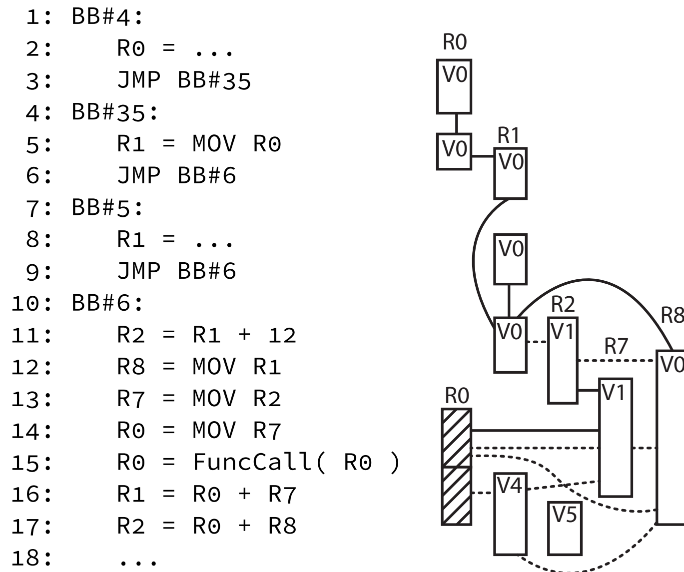


Figure 3.13: Interference graph derived from the register assignment from Figure 3.12. Move related edges are shown as solid lines while interference edges are dotted. The cross-hatch live ranges are fixed physical registers.

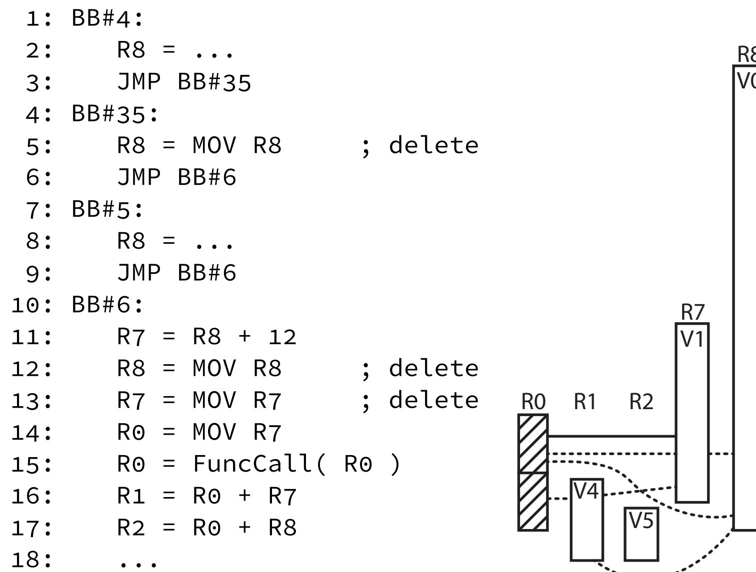


Figure 3.14: Optimized interference graph showing the results of coalescing. Only one required move instruction is left.

placed in the same physical register. In Figure 3.13, interference edges are indicated by a dotted line.

Our construction of the graph, where nodes are bounded by the limits of basic blocks, can lead to a large number of nodes. Many of these nodes, connected by move-related edges, will be in the same physical register. However, we expect it to be more difficult to eliminate move-related edges connected to large nodes than to small ones. By delaying the merging of nodes we hope to make it easier to merge along move-related edges with high execution frequency.

3.6.1 Algorithm

The general algorithm is outlined in the following steps. It begins by building the move interference graph. Graph coloring allocators have received considerable attention since their introduction and have highly optimized data representations. Our algorithm can benefit from this research although we are not implementing a full register allocator. We begin with a valid coloring of the graph and wish to modify it. Since there will be no spilling, the graph will remain consistent and will not need to be rebuilt at any point.

1. Build – Build the Move Interference Graph.
2. Accept – Move related edges between interfering nodes are removed from the graph. They cannot be coalesced.
3. Revive – Promote memory nodes to a register where possible.
4. Simplify – Remove non-move related nodes of low degree. Place on stack.
5. Optimize – Iteratively merge move related edges.
6. Unsimplify – Reinsert simplified nodes by popping off stack.
7. Reduce – Reduce number of physical registers used.

The graph can be represented using a list of nodes, a list of move related edges, and an adjacency matrix that allows quick lookup to determine if two nodes interfere [58]. The node data contains important information about each node such as its register location, number of interferences, and number of move related neighbors.

Some move related edges are between portions of a live range that interfere. This is especially true of move instructions designed to split a live range. As in Figure 3.14, preserving the live range for V1 across a function call while also placing it into an argument register will necessitate at least one move. These move related edges must be accepted by the algorithm and can be removed as a first step.

Revive

We can also allocate nodes representing the regions where a live range is in memory. This makes it possible to revive parts of the live range that are in memory but can be placed in a register where one is available. Reviving parts of the live range can result in eliminating load and store instructions. However, it also increases the number of interferences and may make it more difficult to eliminate move instructions. In practice, we did not find many opportunities to revive parts of a live range nor did we find a significant reduction in spill instruction costs where it was possible.

Simplify/Unsimplify

Simplification of nodes uses the conservative coalescing heuristic of Briggs et al.[22]. Nodes can be removed from the graph if they are of low degree and are not connected by a move related edge. If a node is removed then its assigned physical register may be assigned to any of its neighbors. Once a 'simplified' node is returned to the graph there will be a physical register available to assign to it because it has fewer neighbors than there are physical registers available.

The order that a node is removed from the graph during simplification is important. As nodes of low degree are removed, other nodes may become low degree in the process. When these nodes are added back into the graph during the Unsimplify step they must be added in the reverse order to ensure that a physical register is available for them. A stack data structure is useful for this purpose.

During the Unsimplify step, nodes are removed from the top of the stack and reinserted back into the graph. Upon reinsertion they are assigned an available register which was guaranteed during Simplify.

Optimize Move Related Edges

The optimization stage tries to move the endpoints of a move related edge into the same register location. The approach is greedy in nature as it tries to find easily

1. Sort move related edges by descending weight, most frequently executed first.
2. Iterate through edges and merge those that are easily merged.
   On failure, proceed to step 3.
3. Merge an edge whose endpoints can be forced into a common register.
   On success proceed to step 2.
   On failure, proceed to step 4.
4. Exit

Figure 3.15: Procedure for optimizing move related edges.

merged edges before more difficult ones. Table 3.15 outlines the steps for optimizing move related edges. The edge list, sorted from the most frequently executed to the least, is iterated over from first to last. When merging nodes along a move related edge, a new node is created and the old ones are deleted. The new node assumes a union of the combined interferences of the old nodes as well as the combined move related edges connected to the old nodes.

Two nodes are easily merged along a move related edge, in Step 2, if there is a common empty register that both nodes may be placed into. One, or both, of those nodes may already be in the common register and will not need to be moved. The register locations of the endpoints can be tried first as merging locations. We iterate over each move related edge in the graph until no easily merged edges are found.

Step 3 involves finding a common register by moving other nodes out of it. A node may be forced into a destination register R, if all of its interfering neighbors that are in R can be moved to other registers. This step is expensive because each physical register is a candidate and must be tried.

The node is first removed from its current register which becomes empty. This allows another node to be moved into it. We then sort the physical register candidates by increasing number of interferences in that physical register. For each interference in a physical register we try to move it to another empty register. If a physical register can be made empty for both endpoints of a move related edge then the nodes can be merged into that empty register and the move related edge deleted.

The algorithm could be made recursive in order to do a more exhaustive search for an empty physical register. For each interference that can not be moved, we could try to force that node into a physical register by trying to move its interfering neighbors.

However, in our experience, this is prohibitively expensive and does not significantly improve the results.

Reduce

When coalescing two nodes joined by a move-related edge the algorithm attempts to use any unallocated register for both merging nodes. As a consequence of this, nodes can be spread out over the set of available registers, using more than is necessary. The calling convention can mandate that a function must save a number of these registers, if they are used by the function, so that the contents on function entry can be restored when returning to the caller. By avoiding the use of callee saved registers we can hope to minimize the costs of storing registers to memory.

In this stage, an attempt is made to reduce the number of callee saved registers used by the function. The allocatable registers for the target architecture are ordered with callee saved registers first. This means that all nodes, representing the virtual registers, will consider registers in the same order. We then simply iterate over the list of nodes and attempt to move nodes into a register that appears earlier in the ordering. This may take a few iterations to complete. There are likely more sophisticated techniques but this approach appears to work reasonably well.

Chapter 4

Evaluation

Evaluating a register allocation method can be difficult. Register allocators are complex compiler stages that must solve a number of problems. These include allocation with spilling, spill placement, move coalescing, and assignment of physical to virtual registers. Solving each of these problems is made more difficult when considering the interactions between these stages. The solution to one problem can affect the solution of another. This is also true of the compiler itself, where the allocation for the input code discovered by the register allocator can affect subsequent compiler stages.

The register allocation stage can have two main effects on program code. First, physical registers are assigned to virtual register definitions and uses. Second, spill code and move instructions may be inserted to manage virtual register lifetimes. Both of these can have an impact on execution speed.

Traditionally, register allocation has always been focused on minimizing the number of virtual register spills to memory. Costs associated with spilling are due to the much slower speed of memory compared to the processor. Having to wait for memory to respond, when storing or loading, implies wasted clock cycles that could have been used to execute other instructions. An increase in spills should imply a corresponding decrease in program execution speed.

Unfortunately, as processors have become more complex, the actual effects of spills on program code has become more difficult to assess. Cache memory can blunt the effects of most spills by making temporary storage to memory much faster. It is still possible to "miss" in the cache, forcing the processor to load a value from a slower level of memory. This means that there is no uniform cost to spilling when cache memory is available. Fewer spill instructions does not necessarily mean a proportional increase in execution speed.

Even though spill costs are difficult to determine it is still advisable to avoid spilling to memory. Allocators will try to avoid memory spills by spilling rematerializable virtual registers. These kinds of instructions should be less expensive, in terms of execution speed, than those requiring stores and reloads. Rematerializable virtual registers will have a lower worst case spill cost.

In register allocation, the insertion of move instructions is associated with the notion of code quality. An excessive number of moves can slow execution speed by wasting clock cycles and is, therefore, considered to be of poor quality. These instructions are, in most cases, unnecessary as they indicate a virtual register is not in the register it should be in. While we expect a low count of move instructions during program execution, their impact is normally not as significant as spill instructions.

The general principle of minimizing spill instructions that interact with memory is still worthwhile, even though features of modern processors can reduce the effects of spills on running code. It can be difficult to determine what the exact spill costs will be as these may be dependent on runtime conditions. Fewer spill instructions is still a worthy goal as it means a reduced probability of incurring an execution speed penalty.

4.1 Spill Counts

4.1.1 Static Spill Counts

A static count of spill and move instructions is simply a count of how many instructions of a certain kind there are in the code. Fewer instructions can imply smaller, more compact code. It does not imply faster code because it says nothing about the placement of instructions. A poorly placed spill inside of a loop can lead to the repeated execution of a single instruction accessing memory. However, more compact code may lead to better instruction cache utilization, fewer cache misses, and faster execution times.

4.1.2 Dynamic Spill Counts

The dynamic count of an instruction is a count of how many times that instruction is executed during a run of the program. Dynamic counts provide a better measure of the impacts of instructions on running code than static counts. In terms of register

allocation, the dynamic count of spill instructions provides some indication of the success of the allocator to minimize the impact of spills or push them to less frequently executed regions of the code.

However, dynamic instruction execution counts do not provide the whole story. There is a tendency to believe that spills are equally bad — that individual store and load instructions, for example, will have the same costs associated with delaying the processor while accessing memory. This is not necessarily the case when one considers the mitigating effects of processor features such as store buffering and cache memory. Store instructions need not delay the processor if they are buffered. However, too many in succession can overload that buffer and cause the program to wait until the buffer clears.

Cache memory provides a small amount of very fast memory that is used to mirror frequently used parts of main memory. If a region of memory, such as the stack area, that is used to store spilled virtual registers, is used often enough then it will be kept live and accessible in the cache. This can cut memory access times down to a few clock cycles as opposed to hundreds of cycles or nanoseconds. Spilling or reloading a virtual register will help to keep the stack area in the fastest cache level. This means that there is a possibility that a higher dynamic spill count could lead to faster execution times. However, this is a likely to be a rare occurrence. In general, fewer dynamically counted spills should result in fewer cache misses.

4.2 Implementation

Our allocator was implemented as a separate register allocation pass in version 3.6.2 of the LLVM compiler infrastructure. LLVM is a modern compiler tool-chain in widespread use that supports an extensive set of optimizations. It is designed to support both static and dynamic compilation and, as such, it emphasizes fast compilation time so as not to preclude its use in a just-in-time compiler.

The LLVM Greedy register allocator was designed as a replacement for the previous linear scan algorithm. It uses a priority queue to sort the intervals by decreasing size, allocating longer intervals first. Contrast this with linear scan that allocates in the order that intervals are defined. Allocated intervals can be evicted from a register in favor of another interval if that interval has a higher spill weight. The evicted interval is given another chance at a different register prior to being split into smaller intervals. Splitting an interval can result in the insertion of move instructions, rema-

terializing definition instructions, or inserting spills and reloads. If it is not profitable to split an interval further in order to allocate a register for it then the interval is spilled. The Greedy allocator uses a spill everywhere approach to spill placement. However, these spills will tend to be on smaller intervals.

In contrast to the LLVM allocator, we do not support just-in-time compilation and instead take as much time as needed. However, our compilation times are not excessive. This allows us to spend more time on spill placement and move coalescing. This may afford our allocator an unfair advantage. However, the Greedy allocator is a highly efficient production ready register allocator as evidenced by its use as the preferred allocation method. While it may not be the industry leading method it makes for a worthy comparison.

Another option for implementation was the GNU Compiler Collection (GCC) [2] which is a popular and widely used compiler infrastructure. Its primary focus is on producing executable code and uses a register allocator based on the graph coloring algorithm. The GCC allocator uses a hybrid method that performs local register allocation on basic blocks followed by a global graph coloring method [1, 56]. The local allocator allocates virtual registers whose lifetimes are limited to a single basic block. The global allocator then allocates virtual registers for the entire function. Our local allocation method allocates all virtual registers in a block using the distance information. The GCC allocator may have been a better tool to compare our allocator with since neither is restricted in compilation time. However, we decided against implementing our allocator in GCC due to the expense in development time resulting from the complexity of the code and comparative lack of documentation.

We decided to target the ARM 32 bit RISC architecture. The ARM is one of the most popular processor architectures in the world and provides a simple RISC based load/store memory model. The only instructions that access memory are load, store, and load/store multiple. Contrast this with the Intel architecture that allows memory operands for common instructions such as ADD. This would have complicated the spill decision process since it would have added another option that avoids the use of registers but still has an execution cost.

Our target platform for the experiments was the BeagleBone Black computer that contains an ARM Cortex A8 processor. This processor uses the ARMv7 32-bit architecture with 16 general purpose registers and 32 64-bit floating point registers.

4.2.1 ARMv7 General Purpose (Integer) Registers

Not all of the 32-bit general purpose registers are available for use. The Program Counter (pc) holds the address of the next instruction to be executed. The Stack Pointer (sp) holds the memory address of the temporary storage area for use within a function — primarily for spilled virtual registers. It is adjusted when entering and exiting a function. There is also a Frame Pointer (fp) register that is sometimes used by LLVM and may or may not be available. The Link Register (lr) is used to store the function call return address. It is updated when calling another function with the address of the next instruction to be executed upon return. This register may be used within a function but its value must be saved, incurring a cost, so that the function can correctly return to the caller. This leaves 12 to 14 general purpose registers available for storing 32-bit integer values.

r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	fp	r12	sp	lr	pc

Table 4.1: ARMv7 General Purpose Integer Registers. Caller-save registers are in grey. SP and PC are not available for use.

4.2.2 ARMv7 Floating Point Registers

The Cortex A8 processor also supports floating point arithmetic using a separate Vector Floating Point (VFPv3) unit. There are 32 64-bit floating point (FP) registers available called d0 to d31. The architecture supports registers of other floating point sizes that alias the D registers. There are 32 32-bit FP registers f0 to f31 that alias, or overlap, d0 to d16. There are also 16 128-bit registers q0 to q16 that alias d0 to d31.

<table>
<tr><td colspan="4">q0</td><td colspan="4">q1</td><td colspan="4">q2</td><td colspan="4">q3</td></tr>
<tr><td colspan="2">d0</td><td colspan="2">d1</td><td colspan="2">d2</td><td colspan="2">d3</td><td colspan="2">d4</td><td colspan="2">d5</td><td colspan="2">d6</td><td colspan="2">d7</td></tr>
<tr><td>f0</td><td>f1</td><td>f2</td><td>f3</td><td>f4</td><td>f5</td><td>f6</td><td>f7</td><td>f8</td><td>f9</td><td>f10</td><td>f11</td><td>f12</td><td>f13</td><td>f14</td><td>f15</td></tr>
<tr><td colspan="4">q4</td><td colspan="4">q5</td><td colspan="4">q6</td><td colspan="4">q7</td></tr>
<tr><td colspan="2">d8</td><td colspan="2">d9</td><td colspan="2">d10</td><td colspan="2">d11</td><td colspan="2">d12</td><td colspan="2">d13</td><td colspan="2">d14</td><td colspan="2">d15</td></tr>
<tr><td>f16</td><td>f17</td><td>f18</td><td>f19</td><td>f20</td><td>f21</td><td>f22</td><td>f23</td><td>f24</td><td>f25</td><td>f26</td><td>f27</td><td>f28</td><td>f29</td><td>f30</td><td>f31</td></tr>
<tr><td colspan="4">q8</td><td colspan="4">q9</td><td colspan="4">q10</td><td colspan="4">q11</td></tr>
<tr><td colspan="2">d16</td><td colspan="2">d17</td><td colspan="2">d18</td><td colspan="2">d19</td><td colspan="2">d20</td><td colspan="2">d21</td><td colspan="2">d22</td><td colspan="2">d23</td></tr>
<tr><td colspan="4">q12</td><td colspan="4">q13</td><td colspan="4">q14</td><td colspan="4">q15</td></tr>
<tr><td colspan="2">d24</td><td colspan="2">d25</td><td colspan="2">d26</td><td colspan="2">d27</td><td colspan="2">d28</td><td colspan="2">d29</td><td colspan="2">d30</td><td colspan="2">d31</td></tr>
</table>

Table 4.2: ARMv7 Floating Point Registers. Caller-save registers are in grey.

4.2.3 Calling Convention

The standard ARM function calling convention dictates which registers must be preserved by the function being called. The registers that must be saved by the calling function are highlighted in Tables 4.1 and 4.2. Of the 16 general purpose registers, r4 to r11 and the Stack Pointer register r13 (sp) must be preserved by the function being called (the callee). Registers r0 to r3 are used as function argument and return registers and must be preserved by the caller. They may be overwritten by the function being called. Register r12 is designated as a scratch register which may also be overwritten by the function being called. The Link Register r14 is updated with the function return address prior to entering the function being called. Those registers that may be overwritten (r0-r3,r12,lr) must be preserved by the calling function which can incur additional move or spill instructions.

4.2.4 32-bit Move Instructions

The ARMv7 architecture supports the loading of 32-bit integers using a pair of move instructions. ARM instructions are, generally, 32 bits in size. Integer move instructions, used for the loading of a value into a register, avoid using memory by embedding the integer value into the instruction itself. It is not possible to embed a 32-bit value into a 32-bit instruction so the ARM architecture adds two 16-bit integer move instructions that allow for this. The `movw` instruction loads a 16-bit integer into the lower 16 bits of a register and clears the upper 16 bits. The `movt` moves a 16-bit integer into the upper 16 bits of a register without affecting the lower 16 bits.

We refer to the Mov32 pseudo-instruction in our results that corresponds to the insertion of these two 16-bit move instructions. While the Mov32 is a rematerializable instruction it does have increased cost due to the two instructions being inserted instead of one. Mov32 instructions can also lead to pipeline stalls since both of the corresponding 16-bit move instructions define the same register and must be executed sequentially in `movw`, `movt` order. They cannot be executed in parallel.

4.3 Benchmark

Testing was done using the SPECINT portion of the SPEC2000 benchmark [69]. This benchmark has been superseded by other benchmarks but remains useful. A significant portion of the other register allocation research has been performed using

the SPEC2000 which allows us, at least, a superficial comparison. The benchmark has been designed to exercise the processor while reducing the influence of file input/output. This makes it particularly useful for evaluating our register allocator. The programs included in the benchmark span a range of real-world application areas. The following table outlines the individual benchmarks.

Benchmark	Description	Notes
164.gzip	Data Compression based on LZ77 and Huffman Coding	Five test inputs: graphic, log, program, random, source.
175.vpr	Integrated Circuit Design Combinatorial Optimization	Two test inputs.
176.gcc	GNU C Compiler	Five test inputs: 166, 200, expr, integrate, scilab.
181.mcf	Mass Transit Vehicle Scheduling Network Simplex Algorithm	
186.crafty	Chess Playing Program	
197.parser	Link Grammar Parser for English	
252.eon	Probabilistic Ray Tracer	Three test inputs: cook, kajiya, rushmeier.
253.perlbmk	PERL language compiler and interpreter	Seven test inputs.
254.gap	Groups, Algorithms, and Programming	
255.vortex	Database Transactions	Three test inputs.
256.bzip2	Data Compression based on the Burroughs-Wheeler algorithm	Three test inputs: graphic, program, source.
300.twolf	Computer Aided Design, placement and global routing package for microchip design. Simulated Annealing Algorithm.	

4.4 Results

We tested the code produced by our allocator using both static and dynamic profiling. The static profiling was generated using the Branch Probability and Block Frequency analysis passes built in to LLVM. These passes analyze the structure of the flow graph and the branching conditions and values on which branching depends to estimate those statistics.

Dynamic profiling was performed by running each benchmark to record edge execution counts. An edge between two basic blocks is counted if execution transitions from one to the other. Block frequencies, or how many time execution enters a block, can be derived from edge execution counts by summing the edge counts for those edges entering or exiting the block. This information provides a best-case scenario for our allocator. However, similar information could be available within a just-in-time compilation environment. Profiling code before compilation is entirely possible, especially for embedded applications where just-in-time compilation is too expensive. This information also offers a stable point of comparison as opposed to the rules based static profiling approach that depends on implementation details. The rules or algorithm can change between compiler versions or between compilers.

The results presented include dynamic counts for spill Store and Reload instructions produced by the allocator. These are easily determined as they are labeled in the assembler code produced by LLVM. The values for 32-bit move (Mov32), Rematerializable (Remat), and move register-to-register (MovRR) instructions are dynamic counts of all instructions found in the program. Even though Mov32 instructions are rematerializable we count them separately due to their additional costs. Using them to load a 32-bit value requires two instructions that load each 16-bit half of the value separately. They also define the same register which can cause instruction pipeline stalls if executed one after the other. The Mov32 count is actually a count of each 16-bit move instruction.

Mov32, Remat, and MovRR instruction types are not easily tracked from allocator to output assembler code during compilation. LLVM is not instrumented to do this by default. Tracking instructions can be complicated due to different allocation methods, and the possible transformation or optimization of instructions during subsequent stages of code generation. For Mov32 and Remat instructions, inserting reloads amounts to reissuing the original defining instruction. The original definition can then become redundant if it is no longer required by any use of the virtual register. This makes it difficult to extrapolate the 'added' costs of inserted Mov32 and Remat instructions because some costs, where definitions are deleted, are reduced.

The results also show a comparison of the number of executed instructions (Instrs) for each benchmark. An improvement, or a reduction in the number of instructions executed, is shown by a positive percentage. This implies a reduction in the number of instructions executed and not necessarily the size of the program. While fewer instructions executed can lead to faster execution time, this is not always the case.

Instruction and data cache behavior can play a significant role in execution time along with instruction pipelining, or the ability of the processor to execute instructions in parallel.

Due to the difference in the way Store and Reload are counted compared to the other types of instructions, we have separated the results into two charts for each result.

4.4.1 Dynamic Profiling

In this section we evaluate our Distance based Register Allocator DRA by comparing it to the LLVM Greedy allocator. Our implementation of Bélády's MIN algorithm, using Expected Distance, is an inherently profile guided method. Good profiling information will improve decisions made by the algorithm. As previously described we used dynamically profiled edge counts obtained by a prior run of the benchmarks to direct spill decisions. Block frequency information is used by our spill placement optimization. The LLVM Greedy allocator uses static estimates to guide its spill decisions and placement.

With Distance Multipliers

When using distance multipliers we applied a multiplier of four for 32-bit move pseudo-instructions (Mov32), and 1000000 for other rematerializable instructions that do not require stores or loads. The Mov32 multiplier was chosen based on experience and gives a small preference to Mov32 instructions with a noticeable improvement on store and reload counts. Increasing the multiplier further offers diminishing benefits and can lead to slower execution times due to an increase in the dual 16-bit move instructions.

Figure 4.1 shows the percentage improvement in number of dynamically executed instructions of the DRA allocator over the LLVM allocator. The figure is split into two charts: one showing Stores and Reloads, and the other showing Mov32, Remat, register-to-register moves, and total number of executed instructions. The reason for presenting two charts is that the instruction counting is performed differently between them. The store and reload counts reflect the instructions produced by the allocator while the other chart uses the counts of all instructions of those types in the program, whether produced by the allocator or not.

There is considerable overall improvement in the reduction of Stores and Reloads

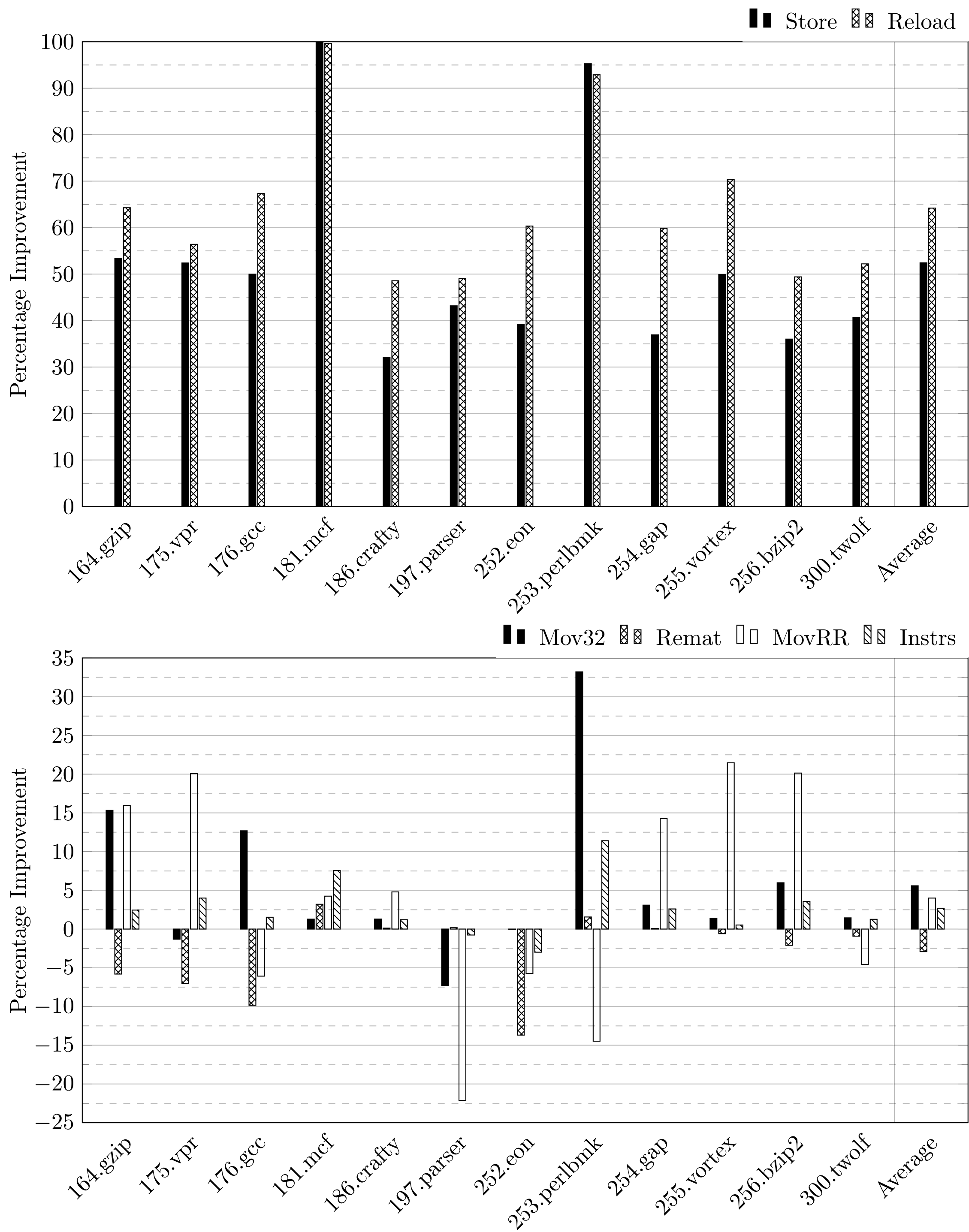


Figure 4.1: Percentage improvement in dynamically executed instructions of the DRA allocator compared to the LLVM Greedy allocator. Using dynamic profiling and distance multipliers.

over LLVM. The DRA allocator executes, on average, 52% fewer Store instructions and 64% fewer Reload instructions.

Although, the Mov32 instruction counts show improvement over LLVM, in general, they appear less dramatic than the counts for stores and reloads. The average improvement, over LLVM, is around 6% fewer Mov32 instructions executed. This is due to the fact that all Mov32 instructions are counted rather than just those produced by the allocator. It also reflects the fact that Mov32 instructions are preferred using the distance multipliers so those virtual registers are more likely to be spilled than those requiring stores and reloads. There is an additional cost to increasing the number of Mov32 instructions so we must be careful not to show too much preference to spilling them.

Rematerializable instructions show an increase in execution count over LLVM of approximately 3%. This indicates a greater number have been spilled by our allocator. This is understandable given the large distance multiplier used in order to influence spill decisions. We find this to be acceptable as it indicates a shift from memory costs, of storing and loading, to the generally lower costs of computing results with operands in registers.

The number of dynamically executed register-to-register move instructions is also improved with around 4% fewer being executed on average. This reflects the bias of our move coalescer to merge along edges with high execution count first.

Finally, the total number of dynamically executed instructions is improved by approximately 2.7% on average. We feel that this is an indication that our DRA allocator is effective at shifting spill instructions to less frequently executed regions of code.

There are outliers in Figure 4.1. The number of Mov32 instruction is significantly worsened for the 197.parser benchmark. However, the increases in the number of Mov32 instructions executed are offset by a reduced number of Store and Reload instructions executed.

There are a few functions in 197.parser with high execution count and having more Mov32 instructions than LLVM. In the worst case, the function `table_pointer` executes over 900 million more Mov32 instructions than LLVM. However, LLVM executes over one billion more Store and Reload instructions, for that same function, than our DRA allocator. So, our allocator has reduced the number of Stores and Reloads by preferring Mov32 instructions which do not access memory and should execute faster. Moreover, the actual number of Mov32 instructions executed is potentially offset by

the number of Store and Reload instructions avoided. The function `region_valid` executes almost 400 million more Mov32 instructions but executes 600 million and 190 million fewer Store and Reload instructions, respectively.

The improvements to Store and Reload instructions are much greater than other benchmarks for 181.mcf. It is a small program with few spills but is also a comparatively long running benchmark. It contains a small but very busy loop in one function of the program. Our store and load results are very good, showing 99.9% and 99.6% improvements in Store and Reload instructions, respectively, compared to LLVM largely because we avoid spilling inside this loop.

A similar situation is evident in the numbers for 253.perlbmk where the improvement in Store and Reload instructions is mostly due to a single function with a very high execution frequency. The function `regmatch` attempts to find the best regular expression rule to the input. It is dominated by a single `switch/case` statement. Distance to next use appears to be a very effective heuristic. Many spill instructions have zero execution frequency. The LLVM Greedy allocator chooses to spill a number of virtual registers where it can place their Store instruction in the entry block. In some cases, this is a good placement as the entry block is likely to have lower execution frequency than loops within the function. However, for functions, such as `regmatch` that are called very often, costs can be high.

With No Distance Multipliers

Figure 4.2 compares the percentage improvement in the number of dynamically executed instructions of the DRA allocator, using dynamic profiling information and no distance multipliers, compared to the LLVM Greedy allocator. Spill choices were, therefore, based on pure distance to next use. The results show that even with no distance multipliers applied, distance-to-next-use works well as a heuristic for making spill choices. However, a good spill placement optimization pass after allocation is expected as spill choices alone will not minimize spill costs.

Reload instructions show positive 29% improvement, on average, over all benchmarks. Store instruction improvements are less certain when no distance multipliers are applied. Without distance multipliers for rematerializable instructions, more virtual registers that require Store and Reloads will be spilled. While the average number of Store instructions executed is a respectable 23% less than the benchmarks compiled using the LLVM allocator, there are several benchmarks showing an increase in the

number of instructions executed over the LLVM benchmarks. The MIN algorithm does not consider store costs when choosing a virtual register to spill so our primary method of reducing the number of Store instructions executed is through the separate spill placement optimization pass. It is apparent, when comparing these results to those with distance multipliers, in Figure 4.1, that the most effective way of reducing store costs is to simply avoid spilling live ranges that must be stored.

Considering the results individually, we see some of the same issues as with the results using distance multipliers. 181.mcf has high Store and Reload improvement due to avoiding spilling inside of a loop of one function in the program. 253.perlbmk shows considerably worse Store results which is a reversal from the positive improvement when using distance multipliers. Once again, this can be attributed to the function `regmatch` that has a very high execution frequency. Although, in this case we suffer from adding two more high cost spills than the LLVM Greedy allocator. Of interest is the considerably better Reload cost than LLVM which can be attributed to the same `regmatch` function.

The percentage improvement for Mov32, Remat, register-to-register move, and total dynamically executed instruction counts is also shown in Figure 4.2. Each of these statistics shows, on average, fewer dynamically executed instructions than LLVM.

Mov32 instructions are somewhat improved for 197.parser over the results when using distance multipliers but are still improved upon over LLVM. There is a small corresponding drop in the number of Store and Reload instructions executed which is largely expected given the shift away from spilling rematerializable instructions.

4.4.2 Static Profiling

Figure 4.3 compares the results for the DRA allocator, when using static profiling information, to the LLVM Greedy allocator. Static profiling is obtained using LLVM's built-in static profiler to generate branch probabilities and block frequency estimates. The DRA allocator uses distance multipliers of 4 for Mov32, and 1000000 for Remat instructions.

With Distance Multipliers

The results are generally positive for both Store and Reload instructions when using distance multipliers as seen in Figure 4.3. Both instruction counts are significantly

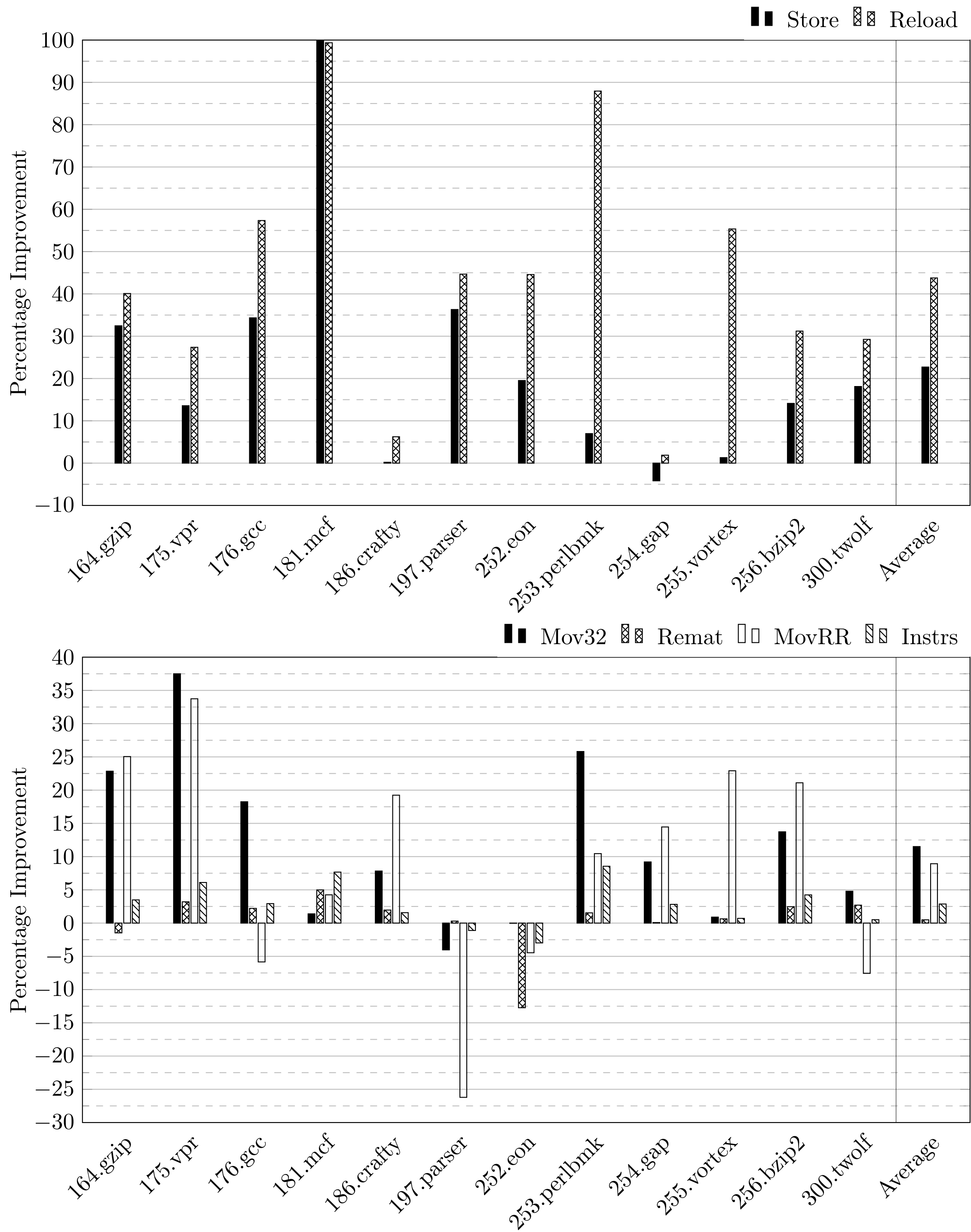


Figure 4.2: Percentage improvement in dynamically executed instructions of the DRA allocator compared to the LLVM Greedy allocator. Using dynamic profiling and no distance multipliers.

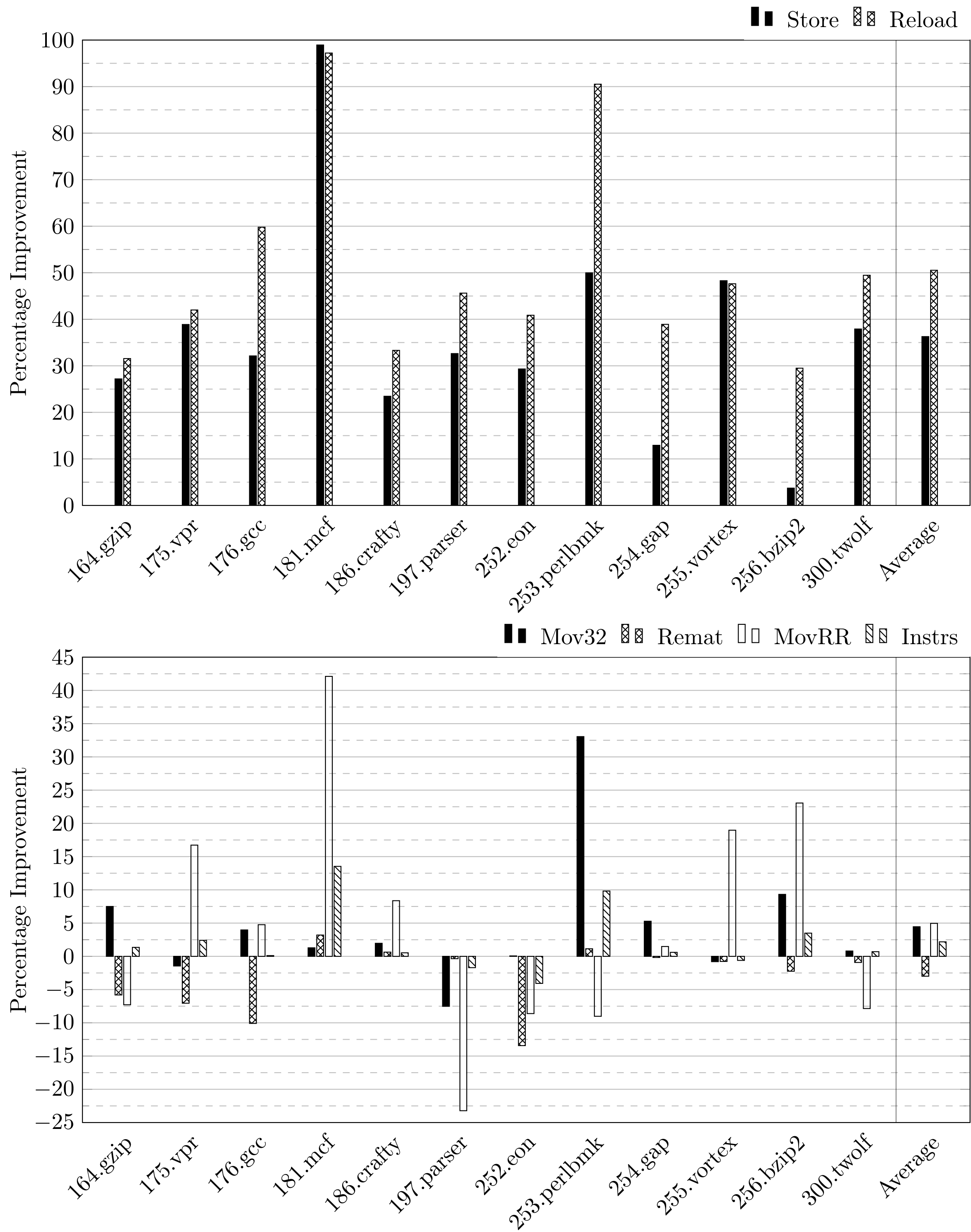


Figure 4.3: Percentage improvement in dynamically executed instructions of the DRA allocator compared to the LLVM Greedy allocator. Using static profiling estimates and distance multipliers.

better than LLVM at 36% and 50% improvements for Stores and Reloads, respectively. These improvements are a decline over those results using dynamic profiling. This is to be expected given that the static profiling information is likely to be less accurate than the dynamic profiling.

The number of dynamically executed Mov32 instructions are reduced by 4.5% on average with most benchmarks showing improvement. The 253.perlbmk benchmark is an outlier showing a large 33% improvement due to good placements within the `regmatch` function that avoids its high execution count. 197.parser shows a decline in Mov32 instructions that is almost identical to the dynamically profiled results. Since the Store and Reload results are not too dissimilar it could be that the static profiling estimates yield branch probabilities that are quite similar to those in the dynamically profiled tests.

Remat instructions appear to follow a similar pattern to those found in the dynamic profiling results for each benchmark. The average increase in number of Remat instructions executed over LLVM is around 3% which is close to that of the dynamically profiled results. We consider this to be acceptable as these instructions are more preferable to spill than the others.

Overall, the results are not quite as good as the previous results using dynamic profiling information with distance multipliers. This is not surprising since the dynamic information is taken directly from a prior program run and is, therefore, more accurate. However, it does show that the static profiling information can be used where dynamic information is unavailable. The accuracy of the static information is important. It seems reasonable to assume that the information provided by LLVM is somewhat accurate based on these results. Should the profiling information be of much poorer quality we might expect the results to be worse.

4.4.3 Nearest Next Use

A previous application of Bélády's MIN algorithm to control flow graphs was implemented using the Nearest Next Use (NNU) by Braun and Hack [20]. Using this heuristic, the distance to next use of a virtual register at a given point in the flow graph is the distance to the next use that is closest on all subsequent paths from that point. The distance is counted as the number of instructions executed to arrive at the next use. Braun and Hack note that edges that exit loops should be assigned a larger distance than a path within the loop. They suggest a length of 100000 be assigned

to loop exit edges.

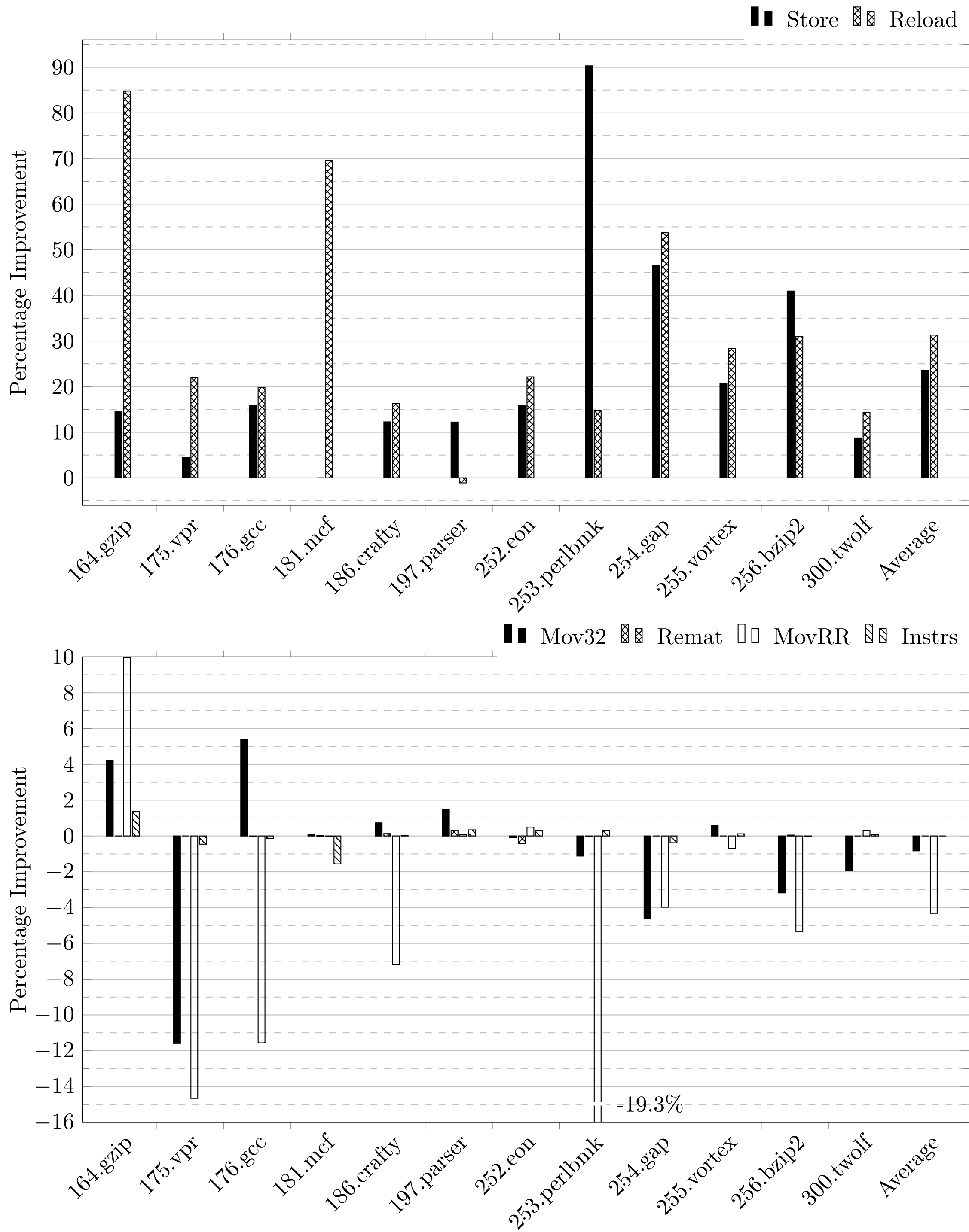


Figure 4.4: Percentage improvement in dynamically executed instructions of the DRA allocator when using Expected Distance over Nearest Next use. Both heuristics use dynamic profiling information and distance multipliers.

The Nearest Next Use is easy to compute on the flow graph as it corresponds to the distance to a particular virtual register use rather than an average over multiple uses. Using iterative data flow analysis it can arrive at a fixed point solution quickly rather than a progressively more accurate solution given enough iterations. It does not use branch probabilities and is therefore immune from the complications of no next use and zero probability edges (see Section 3.3.3).

We compared the Nearest Next Use heuristic to our Expected Distance heuristic by implementing it in the Distance Analysis pass. We were then able to allocate registers with both heuristics using the same method and optimizations. In the following tests, we applied distance multipliers of four for 32-bit move instructions, and 1000000 for rematerializable instructions. We also applied our spill placement optimization to both heuristics to achieve the best results.

An important difference between the two heuristics is that the Expected Distance heuristic uses branch probabilities derived from profiling information in order to determine distance to next use. The Nearest Next Use of a virtual register is not dependent on which path execution follows. However, both heuristics benefit from execution frequencies, derived from profiling information, that are used to guide the spill placement pass.

Dynamic Profiling Figure 4.4 compares the Expected Distance (ED) heuristic to Nearest Next Use (ND) with both using dynamic profiling information. It shows the percentage improvement in dynamically executed instructions of ED over ND. Positive percentages are better.

The chart shows that Expected Distance is clearly more effective at reducing Store and Reload instructions across all of the tests when provided with good profiling information. Only the 197.parser benchmark shows a slight 1% decline. However, on average we find a 24% and 14% improvement in Store and Reload instructions, respectively.

There is a slight overall increase in the number of Mov32 instructions executed by about 0.8%. Although the ED heuristic is better on more benchmarks, the declines are far more significant. Rematerializable instructions were largely the same between the heuristics which probably indicates that they were always chosen to be spilled by both heuristics.

It is difficult to explain the disparity in instructions between the heuristics. Since the ED heuristic appears to produce more Mov32 instructions in some tests, it would

suggest that the ED heuristic chooses Mov32 instructions to spill more often than the ND heuristic. This could, in turn, indicate that Mov32 instructions tend to have uses that are closer to the spill point and thus the ND heuristic seems them as less desirable spill candidates. These Mov32 candidates could also be on less frequently executed paths, making them more desirable to spill, according to the ED heuristic. However, this is a difficult argument to make. The heuristics make different choices at spill points which will alter the sets of spill candidates available to each heuristic at later spill points. This means that each heuristic will, most likely, have different sets of spill candidates to choose from during allocation.

From the results in Figure 4.4, the Expected Distance heuristic is superior to Nearest Next Use when profiling information is quite accurate or reflective of expected branch probabilities. Both Store and Reload dynamic instruction counts were reduced with little overall difference in rematerializable instructions.

Static Profiling Figure 4.5 compares the Expected Distance heuristic (ED) with the Nearest Next Use heuristic (ND) using the default statically derived profiling information in LLVM. The DRA allocator using the Expected Distance heuristic uses branch probabilities to compute distances while both heuristics are aided by the spill placement pass that uses block frequencies.

There is still a significant overall improvement in the number of Reload instructions, by almost 14% using static profiling information. Store instructions also show a slight improvement of more than 3%. However, the 253.perlbmk benchmark is a notable exception showing an almost 76% decline in Store instructions. Once again, this is due to the `regmatch` function in the 253.perlbmk benchmark that contains a switch/case statement. There are many branch targets in a switch statement which can prove problematic when trying to estimate branch probabilities. Without this outlier the Store counts appear to be improved on average.

There is a slight overall improvement in Mov32 instructions of 0.3% indicating that the ED heuristic does not overcompensate for improvements in Store and Reload instructions by spilling Mov32 instructions. The 164.gzip and 175.vpr benchmarks are outliers that largely cancel each other out in the overall results. This is similar to rematerializable instructions which show no observable 0% improvement over the ND heuristic.

There is still an improvement due to the ED heuristic over the ND heuristic when considering spill instructions using static profiling information. There are reductions

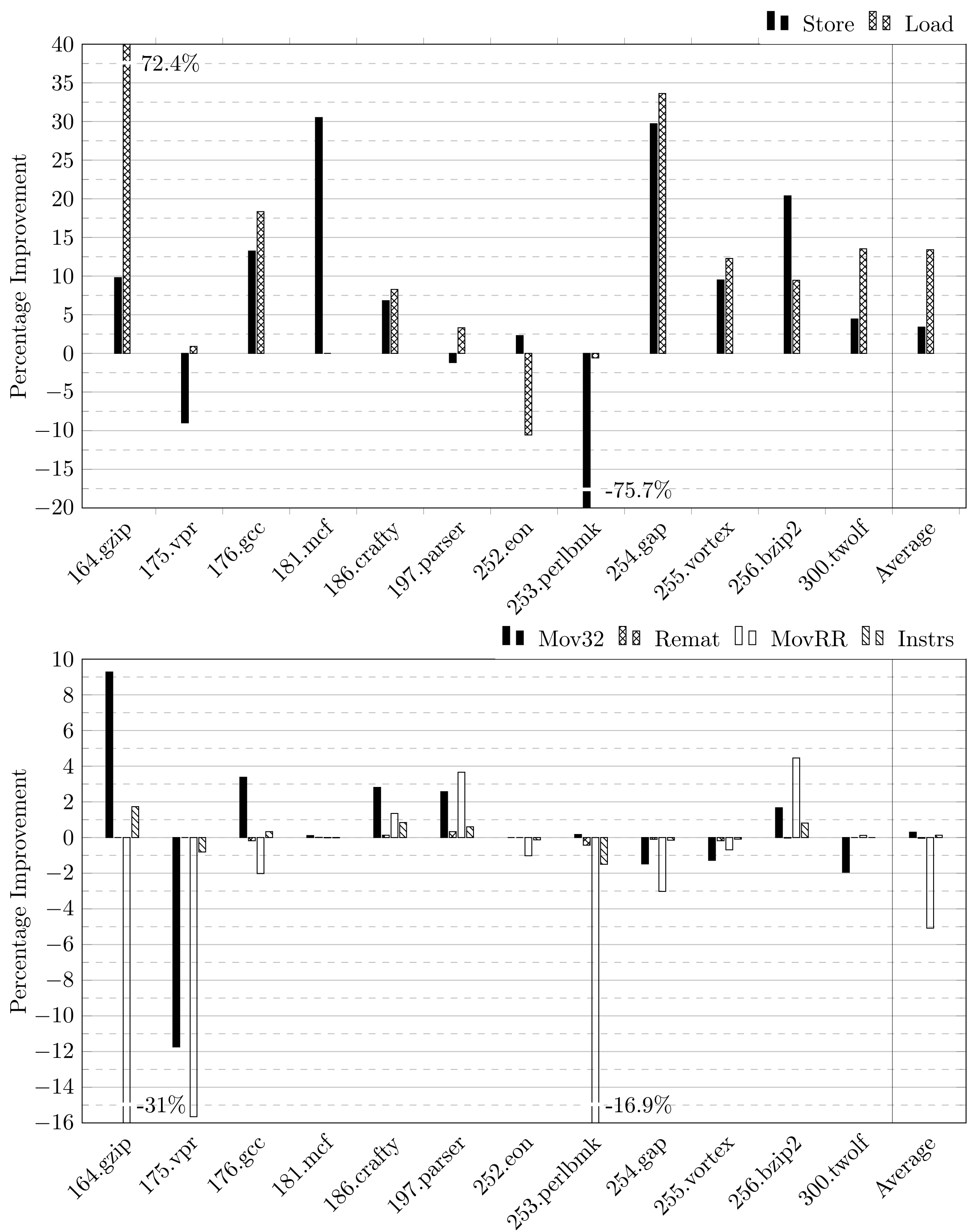


Figure 4.5: Percentage improvement in dynamically executed instructions of the DRA allocator when using Expected Distance over Nearest Next use. Both heuristics use static profiling estimates and distance multipliers.

in the number of dynamically executed Store and Reload instructions without sacrificing Mov32 and Remat instructions in the overall results. Improvements are less significant when shifting from dynamic profiling to static profiling. This underlines the importance of using good branch probability estimates for the Expected Distance heuristic. With good information the Expected Distance heuristic can be used to produce better allocations than the Nearest Next Use heuristic.

4.4.4 Dead Edge Distances

One of the difficulties with computing expected distance is the problem of summing distances across edges when one edge has a zero liveness probability and another does not. The edge with a zero probability is, in effect, a 'dead edge.' We have already discussed possible solutions to this problem in Section 3.3.3. This problem was left unsolved during implementation of the DRA allocator. We simply ignored distances on edges with no next use. However, it is worthwhile to investigate whether estimating distances across dead edges can actually lead to a reduction in spill instruction costs.

We have tested the following estimator, presented as Equation 3.6 in Section 3.3.3, that is only applied at the bottom of a block with multiple outgoing edges where the liveness probability for a virtual register v at the bottom of the block is non-zero $LPBot_B[v] \neq 0$ but at least one of the outgoing edges $B \to S$ the Liveness Probability is zero $LP_{B\to S}[v] = 0$. The reason for this could be because there is no next use across the edge $B \to S$ or because the probability that the edge is traversed is zero $P_{B\to S} = 0$. An example such a situation was shown in Figure 3.4.

$$EDBot_B[v] = \frac{\sum_{(S \in succs(B)) \wedge (LPTop_S[v] > 0)} (P_{B\to S} \times EDTop_S[v] \times LPTop_S[v])}{LPBot_B[v]^2}$$

In all other cases, when all outgoing edges from the block have non-zero distance to next use $(LPTop_S[v] > 0 \wedge P_{B\to S} > 0)$ we apply the normal estimator.

$$EDBot_B[v] = \frac{\sum_{(S \in succs(B)} (P_{B\to S} \times EDTop_S[v] \times LPTop_S[v])}{LPBot_B[v]}$$

What is desirable about this heuristic is that it exponentially scales distance based on the probability that there is no next use at the bottom of the block. This scaling

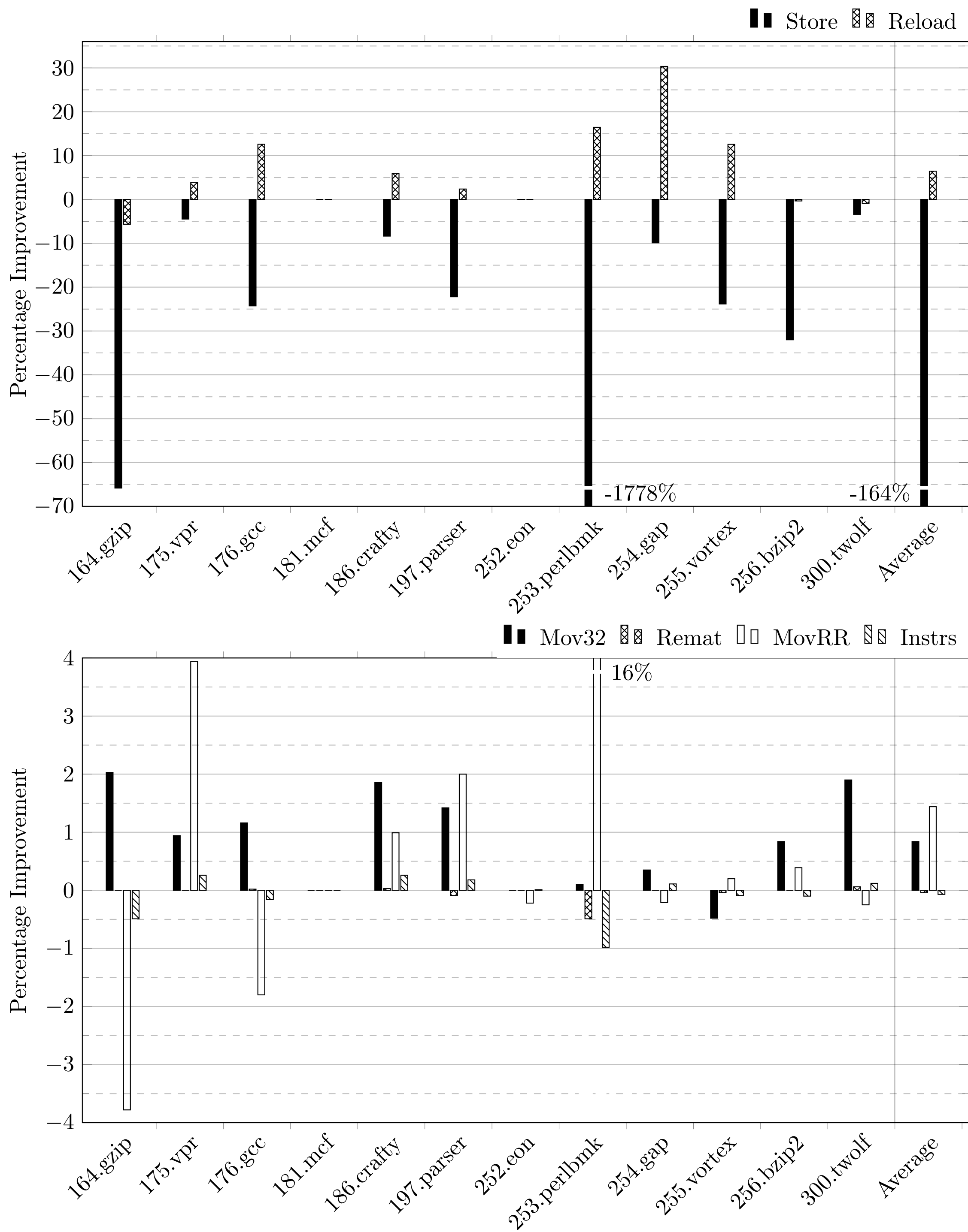


Figure 4.6: Percentage improvement of the DRA allocator when the Dead Edge Distance Estimator is used to compute Expected Distance. The DRA allocator is using dynamic profiling and distance multipliers.

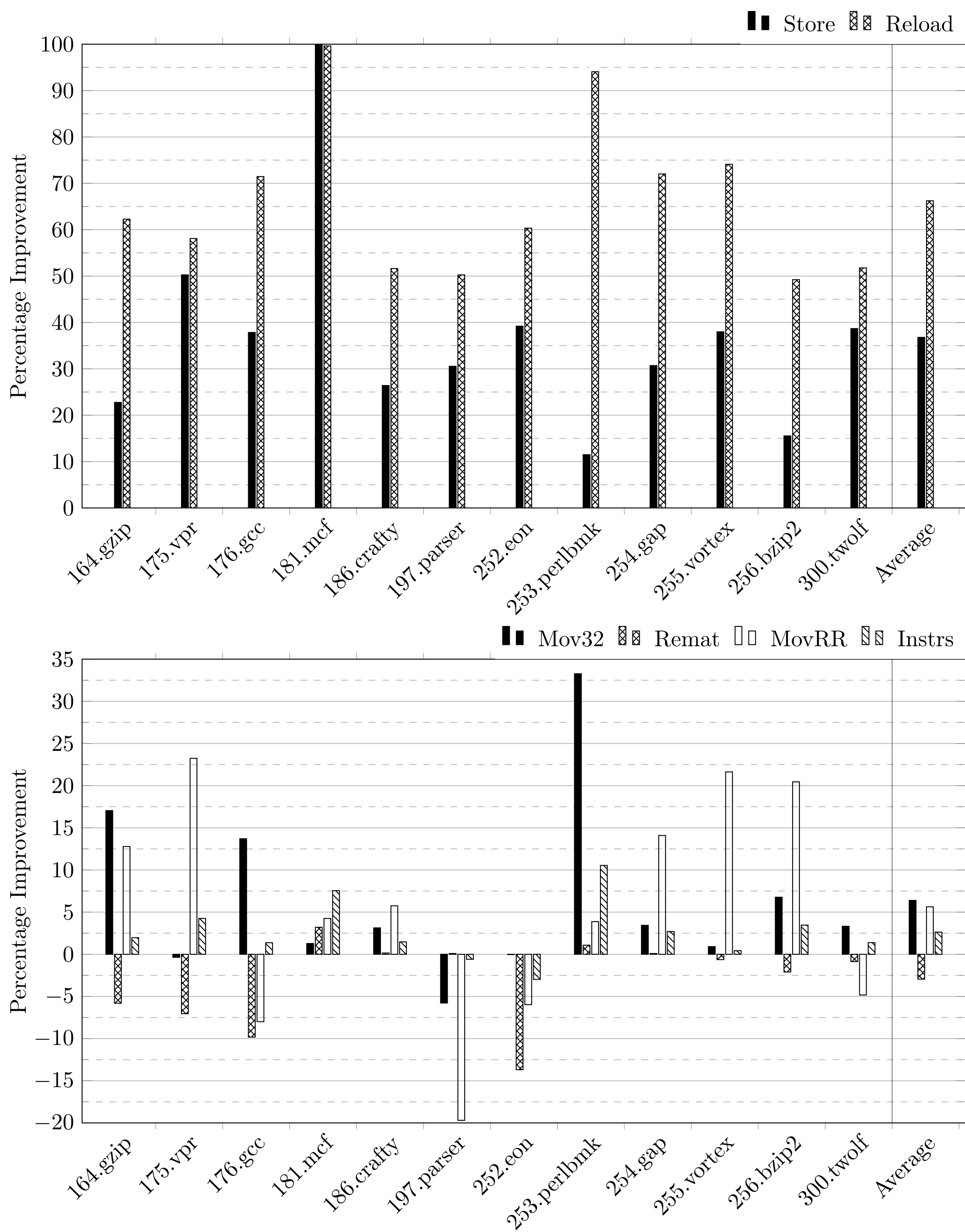


Figure 4.7: Percentage improvement of the DRA allocator, using the Dead Edge Estimator to compute Expected Distance, over LLVM. The DRA allocator is using dynamic profiling, distance multipliers.

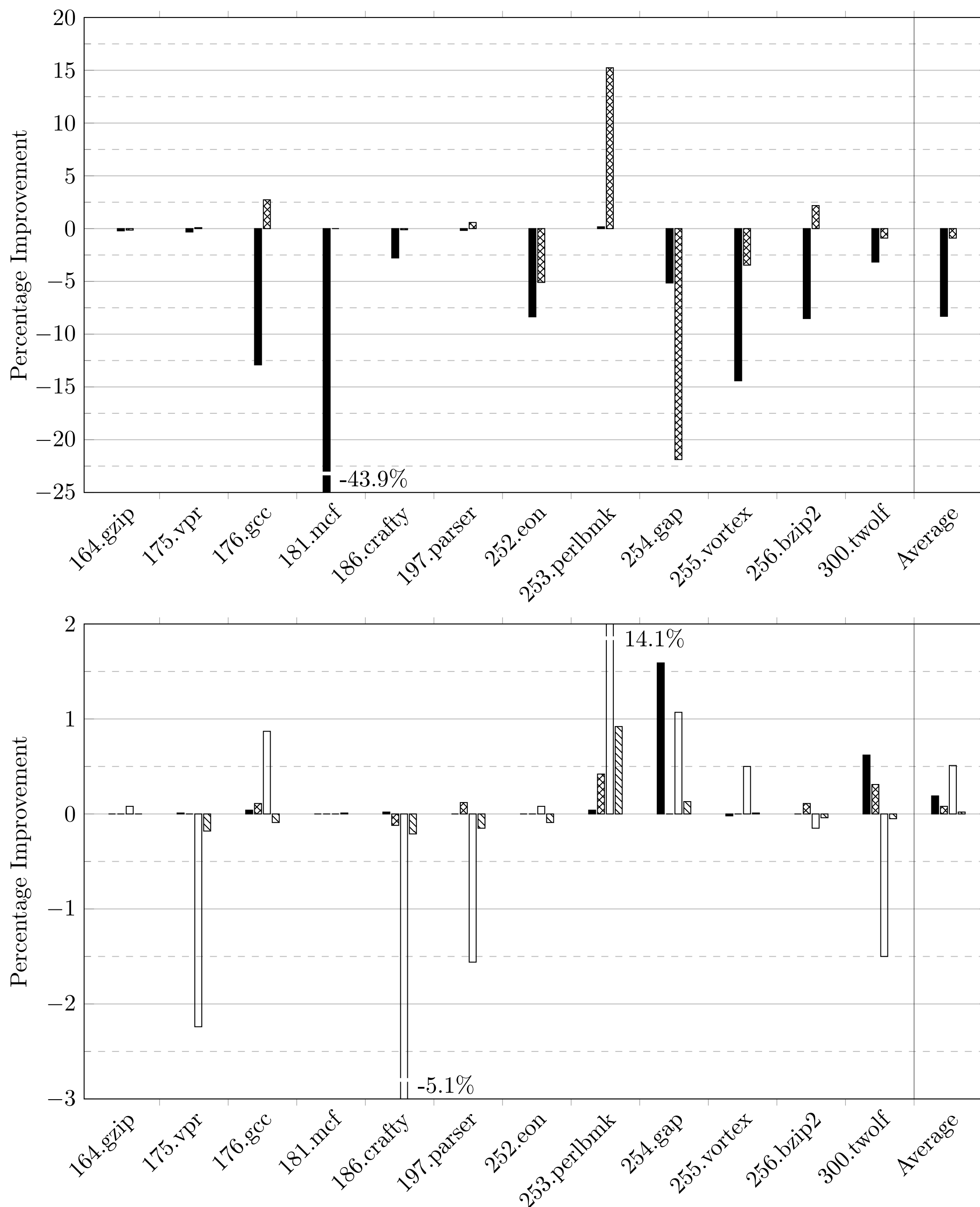


Figure 4.8: Percentage improvement of the DRA allocator when the Dead Edge Distance Estimator is used to compute Expected Distance. The DRA allocator is using static profiling and distance multipliers.

is reasonable without being excessive.

In the figures, we compare the DRA allocator without the dead edge distance

estimator to the DRA allocator with the estimator. The figure, therefore, shows the improvement to allocations when the dead edge distance estimator is not used. Negative numbers imply that the dead edge estimator reduces the execution counts of the respective instruction type.

Dynamic Profiling Figure 4.6 shows the results when the DRA allocator is using dynamic profiling information and distance multipliers of 4 for Mov32 and 1000000 for rematerializable virtual registers. What is most interesting about the results is that both Reload and Mov32 instructions are improved by the dead edge estimator, while Store instructions are made significantly worse. This appears to be most pronounced in the 253.perlbmk benchmark which shows a 1778% decline in dynamically executed Store instructions.

The improvement seen by the dead edge estimator for Reload and Mov32 instructions appears to justify the use of an estimator that takes dead edges into account. However, the significant increase in Store instructions is cause for concern. Store instructions are worse in every benchmark except 181.mcf. We would expect an increase in Store and Reload instructions if the allocator shifts away from spilling rematerializable and Mov32 instructions. Certainly there are fewer Mov32 instructions being spilled, yet there are also fewer Reload instructions being executed. Another possible reason could be that there is less room for movement of store instructions and therefore less opportunity to optimize their placement costs. Reload instruction placement is optimized before Store placement so it is possible that the area in which Store instructions may be placed can become quite constrained.

While the number of Store instructions is increased over the DRA allocator without the Dead Edge Distance Estimator, the numbers still appear favorable when compared to LLVM. Figure 4.7 compares the DRA allocator using the Dead Edge Distance Estimator to LLVM. The number of Store and Reload instructions is improved on each benchmark. Mov32 instructions are improved overall with one exception. These results are largely similar to those of the DRA allocator without the Dead Edge Distance Estimator.

Static Profiling Figure 4.8 compares the use of the Dead Edge Distance Estimator when static profiling and distance multipliers of 4 for Mov32 and 1000000 for rematerializable virtual registers are used. The results are generally poorer with the number of Store and Reload instructions executed appearing worse but with a slight overall

improvement to the number of Mov32 instructions. Rematerializable instructions are also slightly better.

This distance estimator may make the distances more accurate based on the profiling information used. This can lengthen the distances for virtual registers that are non-rematerializable which, in-turn, leads to more non-rematerializable virtual registers being chosen to be spilled. This could explain the increase in Store and Reload instructions executed along with the reduction in Mov32 and rematerializable instructions.

To compensate for the increase in non-rematerializable virtual registers being spilled, the distance multipliers could be adjusted to reduce the number of non-rematerializable virtual registers spilled. However, the multiplier for rematerializable instructions is already quite high so only the Mov32 multiplier should be increased.

From the results, it does not appear that the dead edges heuristic evaluated here can be used to improve spill choices. However, the accuracy of the profiling information becomes even more critical as the difference between the dynamic and static profiling results shows. It is also apparent that the distance multipliers used could be adjusted, for both the dynamic and static profiling results, in an attempt to reduce the spilling of non-rematerializable virtual registers in favor of those defined by Mov32 instructions.

4.4.5 Restricted Registers Experiment

Some architectures have fewer registers available than the ARM architecture. By restricting the number of registers we hope to simulate allocation on these architectures or provide some indication of the ability of our allocator to work with fewer registers. We have, therefore, reduced the number of registers available by six caller saved registers. We have not modified calling conventions or special functions of the remaining registers.

Restricted Register Set Table 4.3 shows the registers available for use. Registers R4 to R6 and R8 to R9 are marked as reserved in the compiler and are not available for use by any compiler stage. Of the rest of the registers, the SP and PC are not available to the allocator as they are required during execution for the stack pointer and program counter, respectively. The FP register can be used as a frame pointer and would be marked as reserved by the compiler if it is used for that purpose. Often

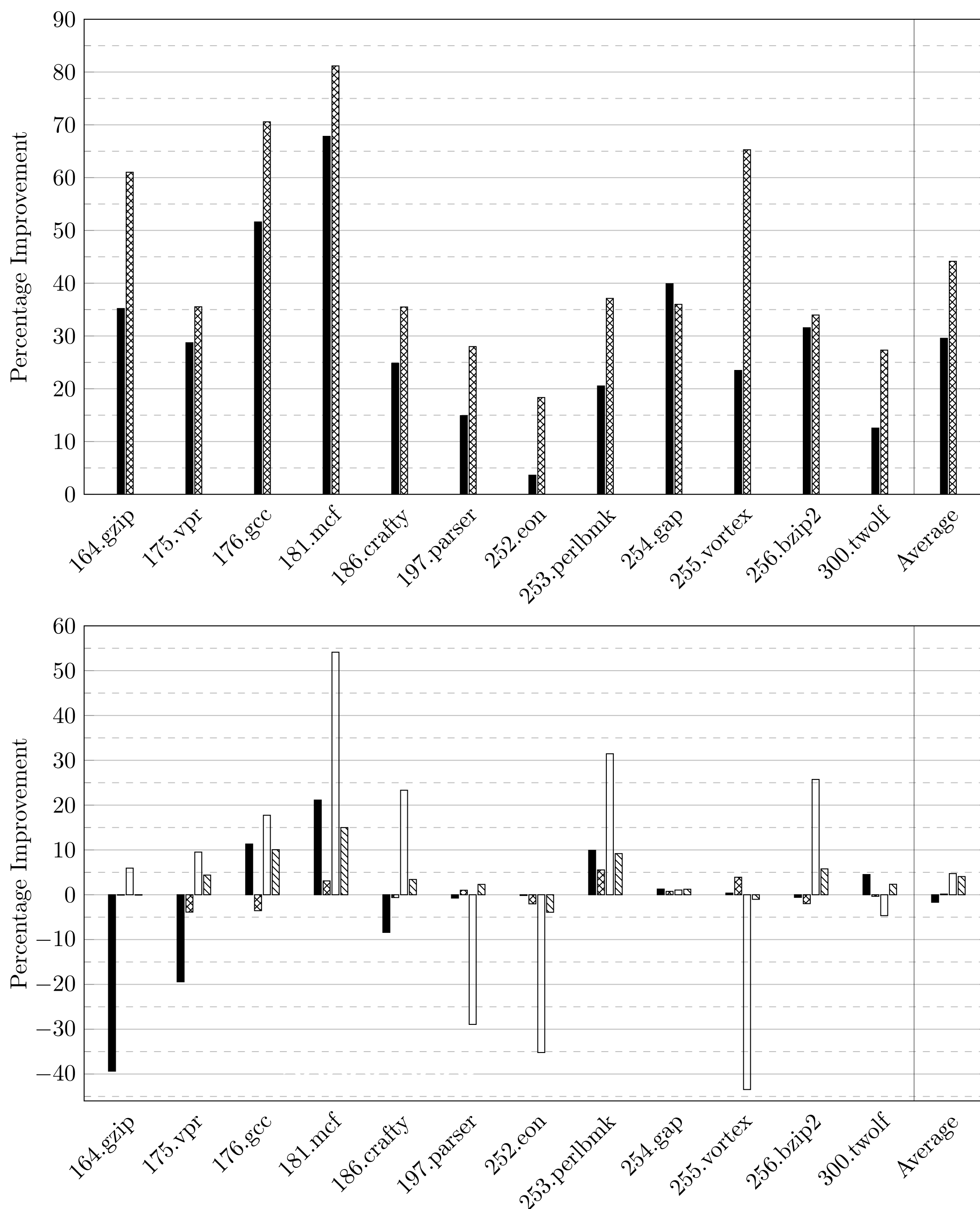


Figure 4.9: Percentage improvement in dynamically executed instructions of the DRA allocator compared to the LLVM Greedy allocator. Both allocators are using a restricted set of registers. The DRA allocator is using dynamic profiling and distance multipliers.

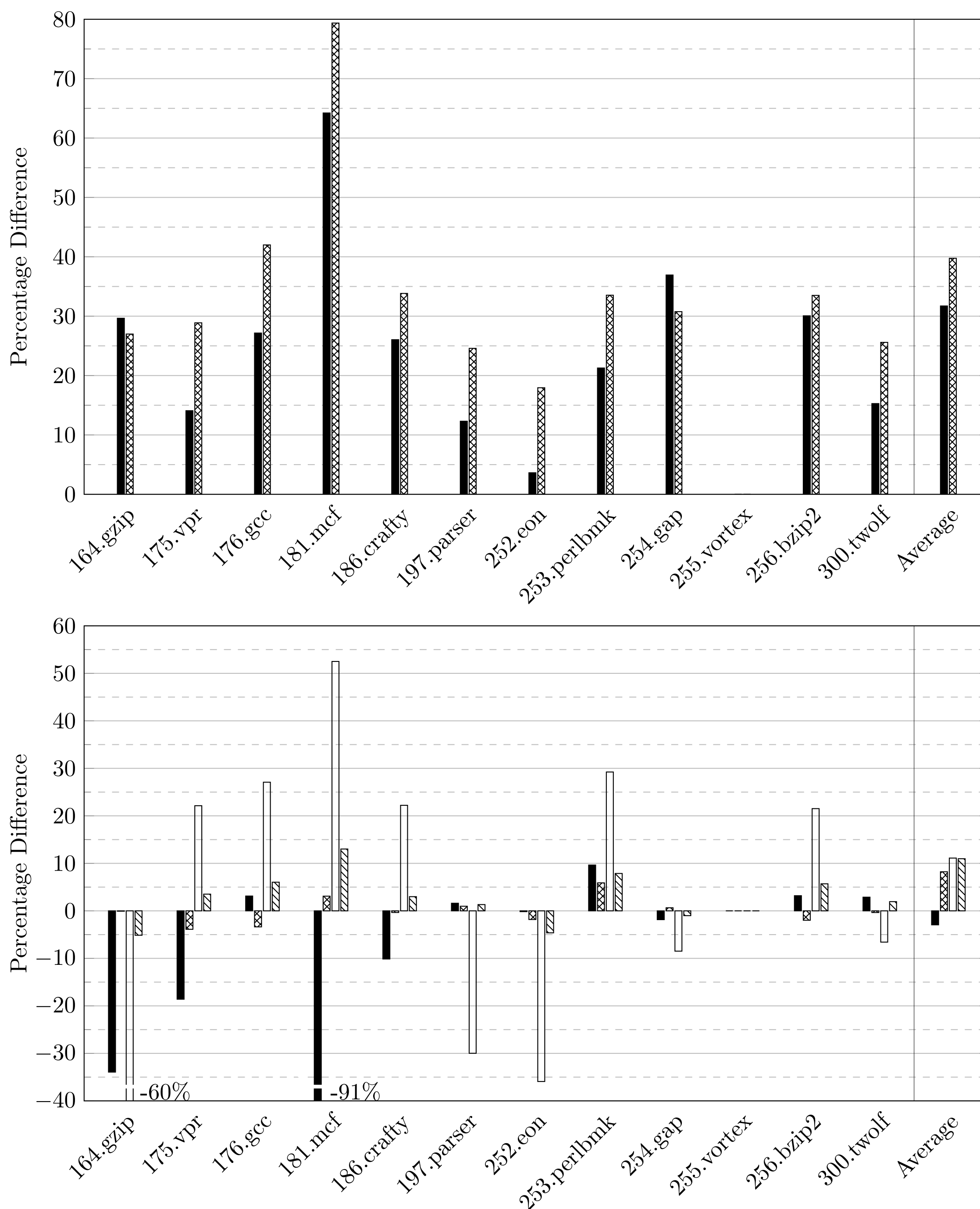


Figure 4.10: Percentage improvement in dynamically executed instructions of the DRA allocator compared to the LLVM Greedy allocator. Both allocators are using a restricted set of registers. The DRA allocator is using static profiling and distance multipliers.

the FP register is available for use by the allocator. Register LR is the link register which is updated by the processor on a Branch-with-Link (BL) instruction that is used to call another function. The LR register can be used for allocation but must be saved on the stack and restored on function exit. Of the remaining registers, R0 to R3 are function argument registers which must be saved by the caller if a function is called. R12 is a scratch register that also must be preserved by the caller.

r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	fp	r12	*sp*	lr	*pc*

Table 4.3: Restricted set of ARMv7 General Purpose Integer Registers. Those marked in grey are not available to any compiler stage including the register allocator.

Figure 4.9 compares the dynamic counts of instructions for the benchmarks between the DRA and LLVM allocators. The DRA allocator is using dynamic profiling and distance multipliers of four for 32-bit move instructions (Mov32), and 1000000 for other rematerializable instructions that do not require stores or loads.

Store and Reload instructions are still significantly better, at 30% and 44% respectively, than the LLVM allocator although not quite as high as the results using the full set of ARM registers in Figure 4.1 which were 52% and 64% respectively. This decline in improvement is not necessarily concerning. We would expect the problem to become more difficult with far more spills than when using the larger set of registers. The fact that we are still considerably better than LLVM with the smaller register set suggests that our method is capable of handling fewer registers.

Mov32 instructions appear to be marginally worse on average than LLVM although two benchmarks show significantly worse results. The 164.gzip benchmark shows a large increase of Mov32 instructions over LLVM. A single function, called `longest_match`, seems to contain a large number of these moves instructions. Unfortunately, the same function also appears to have higher counts for Store and Reload instructions than the LLVM code. The function itself contains a single, very busy, do/while loop which makes allocation difficult. Our allocator appears to spill a loop condition variable whose only use is at the bottom of the loop where costs are very high.

Figure 4.10 compares the dynamic counts due to the DRA allocated code, optimized with static profiling information, with LLVM. The DRA allocator is using the same distance multipliers as the dynamically profiled version. We had some trouble compiling the 255.vortex benchmark so we have zeroed those results.

The DRA allocated benchmarks show a 32% and 40% improvement in Store and

Load instructions, respectively, over LLVM. In comparison, the DRA allocated benchmarks using the full set of registers showed a 36% and 50% improvement in Store and Load instructions, respectively, over LLVM.

In the case of 181.mcf, Mov32 instructions are increased by 91% over LLVM. This is primarily due to a Mov32 instruction placed in a busy loop in a single function. However, while LLVM is far better on the number of Mov32 instructions emitted, it is far worse on Store and Reload instructions for this single function. In fact, the increase in Mov32 instructions executed (562,353,096 more), in the code produced by the DRA allocator, is offset by the decrease in Reload instructions (693,774,794 fewer). Add to this the decrease in Store instructions (1,025,463,340 fewer), and the increase in Mov32 instructions is not as bad as it appears.

Overall, the DRA allocator appears to work well with fewer physical registers available for allocation. While there is improvement on each benchmark for the number of dynamically executed Store and Reload instructions over LLVM, there was a slight decline over the results using the complete set of registers. Spills are far more likely when fewer registers are available yet the DRA allocator is still effective at minimizing spills.

4.4.6 Timings

Improvements to program speed are an important, and often expected, priority for an optimizing compiler. Register allocation should have a direct impact on speed by minimizing the number of spills and reducing their effects when spills are necessary. Spills take time to execute due to the time it takes for memory to respond when a store or load is requested. By maintaining values in registers and avoiding spills, program speed should be maximized.

Figure 4.11 shows the program timings for the DRA allocator, using static and dynamic profiling information, as compared to LLVM. Both versions are using the distance multipliers of 4 $\times$ Mov32 instructions and 1000000 $\times$ Remat instructions.

The timings were recorded for each benchmark running on the BeagleBone Black ARM computer. It uses an ARM Cortex-A8 processor with a dual instruction pipeline, a 32KB L1 data cache, 32KB L1 instruction cache, and a 256KB L2 cache.

The benchmark timings optimized using the DRA allocator with dynamic profiling information showed faster execution times than LLVM in each case except 175.vpr and 300.twolf. Four benchmarks showed greater than 5%, and as much as 10.5%,

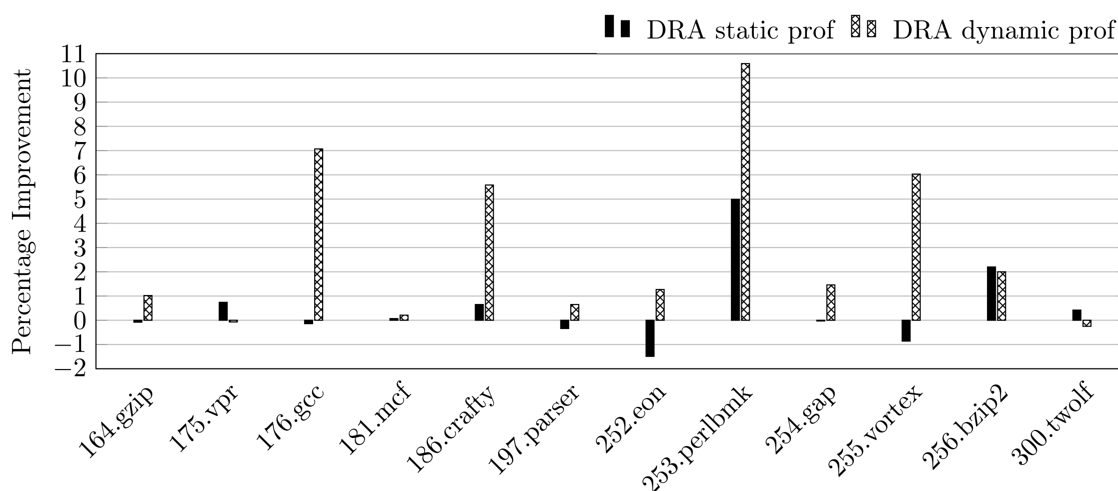


Figure 4.11: Percentage improvement in program running time when compiled using the DRA allocator over code compiled using the LLVM allocator. The DRA allocator is tested with both static and dynamic profiling.

improvement. The remaining benchmark timings were found to be in the region of 0.2 to 2% improvement. These timings were certainly not as dramatic as was hoped for and do not reflect the significant reductions in spill code executed.

Improvements to benchmark timings using static profiling information were not as high as when using dynamic profiling. The results are mixed with significant improvement for both 253.perlbmk and 256.bzip2 but having small declines for 252.eon and 255.vortex.

Hardware Effects on Benchmark Timing

It is clear that when comparing the timings of Figure 4.11 with the reduction in spill code of Figures 4.1 and 4.3 that a reduction in spill instructions executed does not necessarily correspond to a proportional reduction in execution time. This is largely due to the effects of hardware features designed to increase program speed. For the ARM processor used here, the cache memory and instruction pipeline are the primary reasons for, what appears to be, differences in timing.

The L1 data cache offers very fast memory response times, on the order of a few clock cycles, if the requested data is located in the cache. This will almost negate the effects of spills in many cases. What we wish to avoid is a miss in the L1 cache. The second level of cache is significantly slower while having to load from main memory would produce a noticeable slowdown. It is difficult to predict where misses will

occur and is beyond the scope of the register allocator to determine this. The register allocator can only try to minimize spills in order to minimize the potential for cache misses.

The dual instruction pipeline of the ARM processor can execute two instructions per cycle provided that the instructions do not share dependencies. Register read-after-write or write-after-write conflicts will prevent parallel execution. Should a conflict exist between two instructions then one of the pipelines will delay while the first instruction is executed. It is the compiler's responsibility to reorder instructions or renumber registers in order to minimize these conflicts and maximize the use of the dual pipeline.

The 197.parser benchmark showed an increase in 32-bit move instructions in the DRA allocated code over the LLVM allocated code. These can lead to pipeline stalls, due to the two move instruction defining the same register, which may explain the relatively small improvement in timing over LLVM for the code optimized with dynamic profiling and the negative improvement for the code using static profiling. This is not certain, however, because both versions of the DRA code show a high number of move instructions executed relative to the LLVM code which can also slow the benchmark timing.

The LLVM compiler does perform a post-register allocation instruction reordering pass but it will not renumber registers to further improve pipelining. This means that the register allocator has, perhaps, some responsibility to assign registers in such a way as to minimize conflicts. This is difficult to do given that the LLVM compiler performs further optimization after the register allocation pass that could alter some of the register assignments. However, it may be worthwhile to avoid reusing a register immediately after it becomes available and this avoid a write-after-read conflict. It should be noted that, in the ARM code, a significant source of write-after-write conflicts is due to the 32-bit integer move pseudo-instruction which is translated to two 16-bit move instructions that define the same register. An excess number of these can increase the number of pipeline stalls.

Our DRA allocator does not consider reordering when performing assignment. Instead, we try to reduce the number of registers used in each function in the move coalescing pass. By reducing the number of physical registers used we hope to minimize the number that need to be saved on the stack when entering a function and hopefully minimize cache misses. However, this can increase the number of conflicts that lead to pipeline stalls.

a)
```
vcmpe.f32 s2, s0
vldr s0, [r1, #4]
vmrs APSR_nzcv, fpscr
movmi r2, r5
```

b)
```
vcmpe.f32 s2, s0
vmrs APSR_nzcv, fpscr
vldr s0, [r1, #4]
movmi r2, r5
```

Figure 4.12: ARMv7 instruction sequence that a) stalls the processor, and b) its correction.

We also ran into some specific problems with the LLVM compiler. For the 175.vpr results we found that the DRA allocated code, when using dynamic or static profiling, ran on the order of 10 seconds, or 1.5%, slower than the LLVM allocated code. This was due to a single, poorly placed instruction that caused the processor to delay until the floating point unit finished its computation. It seems that the LLVM post-register allocation instruction scheduling pass does not have a rule to handle the placement of these instructions. It was simply unfortunate that the problem appeared in our code and not in the LLVM allocated code.

The code in Figure 4.12 shows the exact problem. The `vmrs` instruction moves the contents of a register in the NEON floating point unit to a core ARM register. In example a), the `vmrs` instruction is allowing the floating point unit to affect control flow by moving the FP condition code register to an ARM condition code register. The `movmi` instruction will transfer `r5` into `r2` if that condition code indicates a negative (minus) condition. Transferring register values between processor units is problematic in ARM. One of the side-effects of the `vmrs` (and the opposite direction `vmsr`) instruction is that it stalls the processor until all floating point instructions have been handled.

This example, taken from the 175.vpr benchmark, is executed 1370219952 times during the program run and results in one of the sub-tests running approximately 10 seconds or 1.5% slower than the reordered code in example b). There is a significant chance that the memory address in r1 is not in the cache so the processor will have to look to the L2 cache or even main memory which can cause significant delays. This is unnecessary as the value loaded by the vldr instruction is not required until later. This appears to be a bug in the LLVM compiler so we moved the vmrs instruction prior to vldr in our timing tests. This allowed the processor to load the value in the background without stalling.

A second problem that we discovered is due to the program layout. The DRA allocated 175.vpr benchmark, using dynamic profiling information, the 175.vpr test

ran considerably slower than its statically profiled counterpart. We found that if we padded the instruction stream at certain points, which would change the locations of functions in memory, we could reduce the execution time by more than 1%. The register allocator may play a role in this problem due to the insertion of spill and move instructions. These instructions affect the size of the functions which, in turn, can affect their locations in memory. However, it is beyond the scope of the register allocator to anticipate this problem, let alone solve it.

Relating Benchmark Timings with Reduction in Spill Code

The DRA allocator appears to do well in functions with large switch statements as compared to LLVM. Profiling information is very helpful when analyzing switch statements as it can be difficult to determine which case statement is the most likely target. For the four fastest benchmarks, when using dynamic profiling information, (176.gcc, 186.crafty, 253.perlbmk, 255.vortex) we found that those functions with a high degree of improvement on spilling had large switch/case statements in them. Sometimes these switch statements were in a while loop or there were combinations of switch/case and if statements.

The 253.perlbmk test shows a better than 10% improvement in execution time. For that benchmark, the improvement may be largely due to good spill choices made in a single function for making regular expression matches. The `regmatch` function consists of a while loop containing a very large switch statement. For the LLVM allocated code, we counted more than five billion Load instructions executed that were due to spills. Both DRA allocated versions execute a mere 2.5 to 3.2% of that number of Loads. We would expect the cache memory to help reduce the effects of spills but, given the high number of Loads executed, the savings in execution time are clearly substantial. It is also worth noting the small difference in improvement between the dynamic and statically profiled versions of the DRA allocated code. This suggests that the allocation method is effective on functions of this kind and is not simply a product of good profiling information.

Long basic blocks should benefit from the use of the Furthest Next Use heuristic. We found one function in the 186.crafty benchmark, called `AttacksTo`, that is a single basic block of more than 130 instructions. We counted 141111930 of both Store and Load instructions executed in the LLVM generated code. The corresponding DRA code has zero spills in that function. The same is true for the statically profiled

version.

Similar results were noticed in the 252.eon code with some functions having very long basic blocks. The LLVM allocator inserted spills into some of those functions while the DRA allocator did not. The timings of the 252.eon benchmark were improved by 1% when using dynamic profiling in the DRA allocator over the LLVM allocated code. While the reduced number of spill instructions executed certainly helped program speed the increase in number of instructions executed over the LLVM produced code may have reduced the benchmark timing improvement. There were significant increases in the number of rematerialized instructions and register-to-register moves.

For some of the benchmarks there is little improvement in timing despite the drastic reduction in dynamically executed spill instructions. As an example, the 181.mcf benchmark exhibited an almost 100% improvement in dynamically executed store and load instructions, when using dynamic profiling, yet only showed a marginal improvement in execution time at 0.07% and 0.2% faster than LLVM. The LLVM allocator made a poor decision to insert a Store instruction inside a while loop that is inside a for loop. The DRA allocator avoided spilling in the same place and was able to move some Store instructions into the entry block.

The 181.mcf benchmark is also one of the longer running programs at around twenty minutes. While we recorded a significant reduction in executed spill instructions these may be only a small part of the overall number of instructions executed. The busiest loops also appear to be small so the costs of spills in the LLVM allocated code may be significantly reduced by the use of cache memory.

The DRA allocator appears to do well on long basic blocks as we had expected. There is also some evidence that the allocator works well on more complex code containing switch statements. The use of profiling information is clearly helpful but it also appears that the DRA allocation method is able to use the profiling information effectively to produce very good allocations.

4.4.7 Summary

It is inherently difficult to relate spill statistics to program execution time. When we see significant reductions in spill instructions executed it is only natural to expect comparable reductions in execution time. However, this impression may be optimistic at best. There are many factors that affect execution time and many of these will

conflict. Different compiler passes may optimize for different goals with one pass degrading the results of previous passes. Significant optimization efforts can even be undone by a single, unpredictable miss in cache memory at runtime.

Our implementation of the DRA allocator makes spill choices and spill placement decisions that are designed to produce low dynamic spill instruction counts, in the general case. We chose parameters for things like the distance multipliers that would work well across all the benchmarks. We do not optimize for specific programs or types of code patterns. For this reason, each program responds differently and results in different levels of improvement for each of the counted instruction types. The results for spill code reduction by our DRA allocator, as compared to the LLVM allocator, are very positive when using both dynamic and static profiling information. We show that our method can win on each of the benchmarks. However, we do not win on each type of instruction. There are increases in the number of Mov32 instructions dynamically executed over LLVM in a small number of the benchmarks. This underlines the fact that it is difficult to improve on every measurement point on all inputs.

It is disappointing to not see a corresponding improvement in program execution time for programs optimized with our DRA allocator as compared to the LLVM produced programs. We would hope to see that the very low number of dynamically executed instructions, in relation to LLVM, would produce significant improvements in execution speed in each case. However, execution speed is affected by many factors. Our register allocation pass is a single pass in a large compiler. Many other passes can help or even hinder the optimizations that our allocator performs. We have observed cases where a later compiler pass, after the allocator, will alter the placement of spill instructions that cause an increase in their execution count. We have also seen cases where the compiler chooses an ordering of instructions that causes the processor to stall which significantly increases execution time. Memory layout and cache misses can also affect execution time. All of these problems are beyond the control of the allocator, yet they play a role in the execution time of the resultant programs, and ultimately in the perception of the effectiveness of the allocation method itself.

We must not expect the allocator to be the main source of reductions in execution time. This is the role of the compiler and not a single pass. The register allocator has the specific task of assigning virtual registers to physical registers while minimizing the impact of spill code and move instructions. It is tempting to want to do more to improve execution speed. Register allocation is inherently tied to memory usage, due to the insertion of spill code, yet the noticeable effects of memory on program

execution speed, such as delays caused by cache misses, are generally unknown until runtime. Therefore, we should avoid increasing the complexity of the register allocator and instead focus on reducing the number of dynamically executed spills as this is the best strategy to avoid using memory and minimizing the likelihood of cache misses due to spills.

Our timing results do not show an execution speed improvement for every program compiled using the DRA allocator when compared to LLVM. What they do indicate is how much improvement one is likely to see when our allocator is used in the compilation of real programs in general use. We are confident in this assertion because the benchmarks that we used are based on real programs and reflect a wide range of behaviors and coding styles. By using the DRA allocator, we can expect speed improvements in 10 programs out of 12 if we have good quality profiling information and the programs being optimized exhibit similar behaviors to the benchmark programs that we used for testing. As the quality of the profiling information decreases so does our expected improvement in program execution time. However, even when using static profiling information we do see improvement in 6 out of 12 of the benchmarks while 3 other benchmarks show an insignificant decline. Our results indicate that our DRA allocator will usually outperform the LLVM allocator and does not perform significantly worse in other situations.

Chapter 5

Conclusion

Register allocation is an important and necessary compiler pass. The goal is to find an assignment of virtual registers, in the program intermediate representation, to the set of physical registers available on the target architecture. Spilling of virtual registers to memory should be avoided, if possible, but may be necessary if too many virtual registers are live at the same time. Minimizing the costs of spills by making good spill choices and optimizing their placement is one of the primary tasks of the register allocator stage.

Since the costs of spills are not realized until a program is executed, register allocators should take into account the run-time behavior of a program when making allocation and spilling decisions. Modern allocators may estimate block execution frequencies, or even support the loading of dynamic profiling information from a previous program run, but these allocators do not fully exploit this profiling information.

We have described a register allocator that uses branch probabilities to make spill decisions and block frequencies to optimize spill placement. Branch probabilities inform the allocator of the most likely execution path and allow it to base spill decisions on which virtual register is least likely to be used next. Block frequencies reveal where the program hot-spots, or regions of high execution frequency, exist and are used by the allocator to minimize potential spill costs.

We are the first to present a method for calculating Expected Distance, using branch probabilities, on the control flow graph and applying it to register allocation. We also describe some of the issues that complicate this calculation. While previous work has presented a Nearest Next Use heuristic for computing distance to next use on control flow graphs, our heuristic provides a more accurate estimate that is more likely to provide a closer approximation to actual distances experienced at

runtime. In fact, our results show that using the Expected Distance heuristic leads to a reduction on Store and Load instructions over using the Nearest Next Use heuristic which reduces Store and Load instructions by 24% and 31%, respectively, with better profiling information.

We have also described our spill placement pass that uses graph contraction to solve the Minimum Cut, and therefore the minimum cost placement of spill instructions, on the control flow graph. While there are other methods that optimize spill placement by solving the Minimum Cut, our method quickly optimizes the placement of both Store and Load instructions. When The spill placement pass is combined with the Expected Distance heuristic, our register allocator proves to be very effective at reducing the number of dynamically executed spill instructions. We have found a 36% and 50% reductions in Store and Load instructions, respectively, when using static profiling information as compared to the competitive LLVM allocator. Results are improved to 52% and 64% reductions in Store and Load instructions when using dynamic profiling information and 36% and 50% reductions when using the less accurate static profiling information.

Execution timings are improved on average by our allocator, showing an almost 3% faster execution speed, on average, when using dynamic profiling information. Unfortunately, these timings are not proportional to the much larger reduction in the number of dynamically executed spill instructions. It would seem that advances in modern processor technology, such as cache memory, may have blunted the effects of spills on running programs. This might suggest that register allocation is less important, as a compiler pass, for optimizing program speed than it may have been in the past. However, we can only partially agree. The register allocation pass remains as important as it has always been in modern optimizing compilers. Minimizing the number of spills emitted by the allocator is, perhaps, the most reliable way to minimize their effects without adversely affecting program speed in the average case. While cache memory can reduce the effects of spills in most cases, inserting more spills is more likely to negatively affect the running times of programs. We believe that the results we have presented here show that aggressively reducing spill instructions will result in improvements in execution time.

5.1 Future Work

- We targeted the ARM processor for our evaluation. It is a Reduced Instruction Set Computing (RISC) processor with specific load and store instructions for memory operations. It would be worthwhile to target a Complex Instruction Set Computing (CISC) processor, such as the Intel x86 architecture, that has a more complex instruction set.

- Distance information is only necessary if spills are forced. We could save some of the computational effort required for computing distance information if we determine where spills are likely to occur. We should only perform the distance computation in these regions of the program and thus avoid unnecessary work.

- Distance multipliers are only applied to rematerializable virtual registers. We did try to apply multipliers to non-rematerializable virtual registers if they were spilled somewhere else in the control flow graph. It may be worthwhile to investigate this further in order to find reasonable multipliers that are based on spill costs. These multipliers would consider whether the virtual register is already spilled on all paths to the spill point, what the store costs could be, and what the load costs might be.

Appendix A

Iteration for Data Flow Analysis

Data flow analysis is the involves computing information about a program on its control flow graph. Often this requires propagating information along edges between basic blocks. For a forward analysis, a basic block will derive its input information from the outputs of its predecessors. A complication to the efficient computation of a data flow analysis is the presence of loops where it is not possible to visit each predecessor of a block prior to visiting the block itself.

Iterative data flow analysis solves this problem by repeatedly visiting the blocks in a function through iteration [46, Chapter 3.4]. If a block has predecessors that have not been visited then that block should be revisited after its predecessors. The general method for solving data flow analysis through iteration is known as the *Round Robin* method. It repeatedly visits the blocks of the function in a set order. This ordering can be forward or reverse and will depend on the data being computed.

Algorithm 8 Iteration limited to N passes over all blocks.

```
procedure IterateA(Blocks)
    count ← N
    while count > 0 do
        foreach block B in Blocks do
            Visit(B)
        count ← count − 1
```

The naive approach to iteration on the control flow graph, shown in Algorithm 8, is to simply iterate over each block a predetermined number of times. The actual number of times is largely a guess based on what the programmer thinks is a sufficient number. This approach is inefficient as the count is likely an overestimation leading

to wasted effort with no improvement to results.

Algorithm 9 Round Robin iterative method that continues while changes are made.

```
procedure IterateB(Blocks)
    changed ← true
    while changed do
        changed ← false
        foreach block B in Blocks do
            if Visit(B) then ▷ Visit returns true if changes made
                changed ← true
```

Iteration should stop when there are no more changes to the result. Algorithm 9 outlines a method of iteration that detects changes to the result. If the analysis reveals that changes have been made to the data for a block then these changes may need to be propagated. The flag "changed" is set which prompts another iteration over the input blocks. Once a complete iteration over all blocks has occurred with no changes being reported, iteration stops.

Care must be taken when detecting changes to the results for a block. The iterative algorithm must be guaranteed to terminate. Each iterative step should converge towards a solution.

The problem with Algorithm 9 is that it unnecessarily iterates over every block in the function. If the inputs to a block have not changed then there should be no need to recompute the results for that block.

Algorithm 10 Iteration using a worklist

```
procedure IterateC(Blocks)
    WorkList ← empty Queue
    foreach block B in Blocks do
        if B has an unvisited predecessor then
            Enqueue(Worklist,B)
        Visit(B)
    while ¬empty(WorkList) do
        B ← Dequeue(Worklist)
        if Visit(B) then ▷ Visit returns true if changes made
            foreach S_B in successors(B) do
                if S_B ∉ Worklist then
                    Enqueue(Worklist,S_B)
```

Algorithm 10 uses a work list to track blocks whose inputs have changed and require updating. This method is far more efficient than the previous method as it

avoids revisiting blocks whose inputs have not changed. The work list may be implemented as a First In, First Out (FIFO) Queue. An initial pass over the blocks in a function detects those blocks that are visited prior to one or more of their predecessors being visited. If this is the case then they may have insufficient information to correctly compute their result. These blocks are placed into the work list so that they may be revisited later.

When servicing the work list, a block is removed and visited. If a change in the result for the block is detected then each successor block is added to the work list (provided it is not already there). Changes are, therefore, more efficiently propagated through the control flow graph.

Bibliography

[1] GCC Wiki: Register Allocation. https://gcc.gnu.org/wiki/RegisterAllocation, 2008. Accessed: December 2, 2015.

[2] GCC, the GNU Compiler Collection. http://gcc.gnu.org/, November 2015. December 2, 2015.

[3] Andrew W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33(4):17–20, 1998.

[4] Andrew W. Appel and Lal George. Optimal spilling for cisc machines with few registers. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 243–253, New York, NY, USA, 2001. ACM.

[5] Andrew W. Appel and Jens Palsberg. *Modern compiler implementation in Java.* Cambridge University Press, New York, NY, USA, 2 edition, 2002.

[6] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[7] John Aycock and R. Nigel Horspool. Simple generation of static single-assignment form. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 110–124, London, UK, 2000. Springer-Verlag.

[8] John Backus. The History of FORTRAN I, II, and III. In Richard L. Wexelblat, editor, *History of Programming Languages I*, pages 25–74. ACM, New York, NY, USA, 1981.

[9] Thomas Ball and James R. Larus. Branch prediction for free. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 300–313, New York, NY, USA, 1993. ACM.

[10] Thomas Ball and James R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[11] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Commun. ACM*, 12(6):349–353, 1969.

[12] L.A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[13] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[14] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill code minimization via interference region spilling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 287–295, New York, NY, USA, 1997. ACM.

[15] D. Bernstein, M. Golumbic, y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 258–263, New York, NY, USA, 1989. ACM.

[16] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register Allocation: What Does the NP-completeness Proof of Chaitin Et Al. Really Prove? Or Revisiting Register Allocation: Why and How. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, LCPC'06, pages 283–298, Berlin, Heidelberg, 2007. Springer-Verlag.

[17] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 102–114, Washington, DC, USA, 2007. IEEE Computer Society.

[18] Florent Bouchez, Alain Darte, and Fabrice Rastello. Advanced conservative and optimistic register coalescing. In *Proceedings of the 2008 international conference*

on Compilers, architectures and synthesis for embedded systems, pages 147–156. ACM, 2008.

[19] Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Trans. Program. Lang. Syst.*, 16(6):1684–1698, November 1994.

[20] Matthias Braun and Sebastian Hack. Register spilling and live-range splitting for ssa-form programs. In *Proceedings of the 18th International Conference on Compiler Construction: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, CC '09, pages 174–189, Berlin, Heidelberg, 2009. Springer-Verlag.

[21] Matthias Braun, Christoph Mallon, and Sebastian Hack. Preference-guided register assignment. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 205–223, Berlin, Heidelberg, 2010. Springer-Verlag.

[22] P. Briggs, K.D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, 1994.

[23] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.*, 28(8):859–881, July 1998.

[24] Preston Briggs, Keith D. Cooper, and Linda Torczon. Coloring register pairs. *ACM Lett. Program. Lang. Syst.*, 1(1):3–13, 1992.

[25] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 311–321, New York, NY, USA, 1992. ACM.

[26] Qiong Cai and Jingling Xue. Optimal and efficient speculation-based partial redundancy elimination. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 91–102, Washington, DC, USA, 2003. IEEE Computer Society.

[27] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 192–203, New York, NY, USA, 1991. ACM.

[28] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 98–105, New York, NY, USA, 1982. ACM.

[29] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages 6*, pages 47–57, 1981.

[30] Chandra S. Chekuri, Andrew V. Goldberg, David R. Karger, Matthew S. Levine, and Cliff Stein. Experimental study of minimum cut algorithms. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '97, pages 324–333, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[31] Keith Cooper, Anshuman Dasgupta, and Jason Eckhardt. Revisiting graph coloring register allocation: A study of the Chaitin-Briggs and Callahan-Koblenz algorithms. In *In Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, 2005.

[32] Keith D. Cooper and Anshuman Dasgupta. Tailoring graph-coloring register allocation for runtime compilation. In *Proceedings of the 2006 International Symposium on Code Generation and Optimization (CGO'06)*, 2006.

[33] Keith D Cooper, Timothy J Harvey, and Linda Torczon. How to build an interference graph. *Software-Practice and Experience*, 28(4):425–44, 1998.

[34] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[35] Dietmar Ebner, Bernhard Scholz, and Andreas Krall. Progressive spill code placement. In *CASES '09: Proceedings of the 2009 international conference on*

Compilers, architecture, and synthesis for embedded systems, pages 77–86, New York, NY, USA, 2009. ACM.

[36] Martin Farach and Vincenzo Liberatore. On local register allocation. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 564–573, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[37] M. R. Garey and D. S. Johnson. The complexity of near-optimal graph coloring. *J. ACM*, 23(1):43–49, January 1976.

[38] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18:300–324, May 1996.

[39] Stephen Guattery and Gary L. Miller. A contraction procedure for planar directed graphs. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '92, pages 431–441, New York, NY, USA, 1992. ACM.

[40] Jia Guo, Maríá Jesús Garzarán, and David Padua. The Power of Belady's Algorithm in Register Allocation for Long Basic Blocks. pages 374–389, 2003.

[41] Sebastian Hack and Gerhard Goos. Optimal register allocation for SSA-form programs in polynomial time. *Information Processing Letters*, 98(4):150–155, May 2006.

[42] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index register allocation. *J. ACM*, 13(1):43–61, 1966.

[43] Wei-Chung Hsu, Charles N. Fisher, and James R. Goodman. On the minimization of loads/stores in local register allocation. *IEEE Trans. Softw. Eng.*, 15(10):1252–1260, 1989.

[44] David R. Karger. Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 21–30, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[45] David R. Karger and Clifford Stein. A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640, July 1996.

[46] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[47] David Koes and Seth Copen Goldstein. A progressive register allocator for irregular architectures. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 269–280, Washington, DC, USA, 2005. IEEE Computer Society.

[48] David Ryan Koes and Seth Copen Goldstein. Register allocation deconstructed. In *SCOPES '09: Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems*, pages 21–30, New York, NY, USA, 2009. ACM.

[49] Priyadarshan Kolte and Mary Jean Harrold. Load/store range analysis for global register allocation. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 268–277, New York, NY, USA, 1993. ACM.

[50] Timothy Kong and Kent D. Wilken. Precise register allocation for irregular architectures. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 297–307, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[51] A. Koseki, H. Komatsu, and T. Nakatani. Spill code minimization by spill code motion. In *12th International Conference on Parallel Architectures and Compilation Techniques (PACT 2003)*, pages 125–134, Sept 2003.

[52] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[53] Jonathan K. Lee, Jens Palsberg, and Fernando Magno Quintão Pereira. Aliased register allocation for straight-line programs is NP-complete. *Theor. Comput. Sci.*, 407(1-3):258–273, 2008.

[54] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steven Tjiang, and Albert Wang. Storage assignment to decrease code size. *ACM Trans. Program. Lang. Syst.*, 18(3):235–253, May 1996.

[55] Vincenzo Liberatore, Martin Farach-Colton, and Ulrich Kremer. Evaluation of algorithms for local register allocation. In *CC '99: Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, pages 137–152, London, UK, 1999. Springer-Verlag.

[56] Vladimir N Makarov. Fighting register pressure in GCC. In *GCC Developers' Summit*, page 85, 2004.

[57] Hanspeter Mössenböck and Michael Pfeiffer. Linear scan register allocation in the context of SSA form and register constraints. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 229–246, London, UK, 2002. Springer-Verlag.

[58] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[59] Jens Palsberg. Register allocation via coloring of chordal graphs. In *CATS '07: Proceedings of the thirteenth Australasian symposium on Theory of computing*, pages 3–3, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[60] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, 26(4):735–765, 2004.

[61] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation after classical SSA elimination is NP-complete. In *Proceedings of Foundations of Software Science and Computation Structures*, volume 3921 of *Lecture Notes in Computer Science*, pages 79–93, March 2006.

[62] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation. In *In Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 109–121. ACM Press, 1997.

[63] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999.

[64] Todd A. Proebsting and Charles N. Fischer. Demand-driven register allocation. *ACM Trans. Program. Lang. Syst.*, 18(6):683–710, 1996.

[65] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation by puzzle solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 216–226, New York, NY, USA, 2008. ACM.

[66] F. Rastello, F. de Ferrière, and C. Guillon. Optimizing Translation Out of SSA Using Renaming Constraints. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 265–, Washington, DC, USA, 2004. IEEE Computer Society.

[67] Vivek Sarkar and Rajkishore Barik. Extended linear scan: An alternate foundation for global register allocation. In *Compiler Construction, 16th International Conference, CC 2007*, pages 141–155, 2007.

[68] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288, New York, NY, USA, 2004. ACM.

[69] 2000. Standard Performance Evaluation Corporation. SPEC CPU2000 benchmark suite, https://www.spec.org/cpu2000/., 2000.

[70] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, July 1997.

[71] Mikkel Thorup. Structured programs have small tree-width and good register allocation. *Information and Computation*, 142:318–332, 1995.

[72] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 142–151, New York, NY, USA, 1998. ACM.

[73] Christian Wimmer and Michael Franz. Linear scan register allocation on SSA form. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 170–179, New York, NY, USA, 2010. ACM.

[74] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 132–141, New York, NY, USA, 2005. ACM.

[75] Jingling Xue and Qiong Cai. A lifetime optimal algorithm for speculative PRE. *ACM Trans. Archit. Code Optim.*, 3(2):115–155, June 2006.