Architecture Independent Register Allocation

by

Felix Stanley Klock II

B.S., Massachusetts Institute of Technology (2000)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2001

© Felix Stanley Klock II, MMI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 3 1 2002
LIBRARIES
BARKER

Author .....
Department of Electrical Engineering and Computer Science
August 3, 2001

Certified by .....
Martin C. Rinard
Assistant Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Architecture Independent Register Allocation

by

Felix Stanley Klock II

Submitted to the Department of Electrical Engineering and Computer Science
on August 3, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Electrical Engineering

Abstract

This thesis presents a framework for generic register allocation, in which the allocation algorithm is completely seperated from the constraints of the source language or target architecture. This allows the backend of a compiler to be designed in a modular fashion: one module implements the allocation algorithm, while another module describes the properties of the targetted architecture. The design of the framework is described using the Alloy modelling language, in order to illustrate the approaches used to handle various issues that arise when attempting to abstractly represent the constraints introduced by the assembly code processed by the allocator. The framework is one component of the MIT Flex compiler infrastructure, which is written in Java and compiles Java bytecodes into a variety of target assembly languages, such as MIPS and ARM.

Thesis Supervisor: Martin C. Rinard
Title: Assistant Professor

Acknowledgments

I am grateful to my advisor Martin Rinard for his guidance and support throughout my time at the Laboratory for Computer Science. Thanks also to the other members of the MIT Flex group for their work on the many components of infrastructure. Special thanks to Emmett Witchel for his work on the MIPS compiler backend, which turned the theory of this project into practice. Finally, many thanks to C. Scott Ananian for his hard work on all aspects of the Flex compiler, from the Java bytecode front-end to the runtime support libraries (and all the bits in between), and additionally for being a constant source of advice, practical and frivolous, and amusement in our office over the past two years.

Contents

1 Introduction **9**

2 General Register Allocation **15**

2.1 Alloy Model of Computation 16

2.1.1 Control Flow 17

2.1.2 Variable References 18

2.2 Modelling Register Assignment 19

2.3 Graph Coloring Register Allocation 20

2.3.1 Spilling 20

2.3.2 Rewriting Code 22

2.3.3 Coalescing 23

2.4 Related Work 24

2.4.1 Other Generic Register Allocators 25

2.5 Summary 26

3 Handling Assignment Constraints and Multiword Temps **27**

3.1 Register Types 27

3.2 Register Clusters 28

3.2.1 Problem Analysis 30

3.2.2 Nickerson's Approach to Register Clusters 30

3.2.3 Briggs' Approach to Register Clusters 31

3.3 Joint-Weighted Graphs 33

3.3.1 An Unexpected Benefit of Pressure 37

3.4 Mapping Temps to Assignments 37
3.5 Summary 38

4 Handling Cross Instruction Assignment Constraints 39
4.1 Conditional Execution on the StrongARM 39
4.2 Expansion into Multiple Instructions 40
4.3 Assembly String Coalescing 41
4.4 Cross Instruction Constraints 42
4.5 Dummy Instrs and Spilling 44
4.6 Beyond Assembly String Coalescing 45
4.7 InstrGroups 46
4.8 Summary 48

5 Implementation within the Flex Compiler Infrastructure 49
5.1 Modules of the Flex Compiler 49
5.2 Instr form 50
5.3 InstrGroups 51
5.4 Code Generation 51
5.5 Register Allocation 52

6 Experimental Results 53

7 Future Work and Conclusions 57
7.1 Opportunities for Future Research 57
7.2 Possible Drawbacks of Genericity 58
7.3 The Coalesced Object Model 59
7.4 Conclusions 59

A Register Allocation Interface 61

Chapter 1

Introduction

Computer systems research has introduced numerous processor architectures over time. While the majority of the consumer desktop PC market has settled on the Intel 80x86 family as a standard, there is still a wide variety of processors in use in and being deployed for servers and embedded systems.

This places a massive burden on compiler developers seeking to properly support every possible target architecture requested by client users. Software engineering discipline dictates that the majority of the code in a compiler should be architecture independent; that is, developers should specify and write algorithms in a generic fashion so that they can be reused when targeting as many computer systems as possible. Such reuse, if carried out properly, yields systems that are scalable, easily maintainable, and easily understandable.

Many compilers are designed with just such a modular structure in mind, with a front-end that parses the input language, converting it into an intermediate representation (IR) of the program being compiled, passes that perform analysis and transformations on the intermediate code, and a back-end that converts the intermediate code into the language of the target architecture (usually assembly code). This structuring allows for relative ease in extending the compiler to support new input languages and to target new architectures.

One key component of a compiler is the register allocator, which is responsible for controlling the processor's maintenance of temporary variables during execution.

However, many register allocation algorithm implementations are not written generically. Instead, they depend implicitly upon the computer architecture being targeted; such a dependence seems natural, since the register file is a processor-specific component.

Tying the allocator implementation to the compiler backend of the compiler forces the allocation algorithm to be rewritten for every targeted architecture. This:

- Increases the chance that bugs will be introduced to the register allocation pass
- Complicates experimentation with new register allocation algorithms or heuristics, since each allocator is likely to have architecture specific extensions strewn throughout the code base
- Increases the size of the maintained code base
- Makes it difficult for one backend to benefit from the enhancements made to another backend's register allocator

Additionally, the abstract problem of register allocation can be specified in an architecture independent fashion. The essential problem is to provide an algorithm that, given a program, yields an (Instruction $\times$ Variable) $\mapsto$ Register mapping[1]. There are three main problems with this simplistic view of a register allocation implementation:

- Many architectures provide different register banks for holding different types of values (for example, a bank of floating point registers versus integer registers)
- Some source programs declare variables that hold values that cannot fit into a single register for a specific architecture (for example, the type "long" in C often will not fit into a single register [5])
- Several architectures constrain how their registers can be used (for example, the StrongARM requires the first two operands to the `mul` instruction to be distinct)

[1]Intuition dictates that a Variable $\mapsto$ Register mapping would suffice, but that could significantly constrain the allocation, as it would not allow for a variable to be stored in different registers in different contexts

All three of these problems essentially stem from the fact that the so-called "General Purpose Register" is a mythical beast; in most modern architectures, registers are isolated into specific groups as a processor level performance enhancement.

A framework for generic register allocation can solve the above problems by defining a way to abstractly express the constraints introduced by the source language and the target architecture, and then establishing an interface through which an architecture dependent module communicates the constraints to an architecture independent register allocator. We present one such framework here.

Such a framework provides many advantages.

- It allows for easier development of new compiler backends, since the total amount of code that must be devoted to the new backend is smaller, and the considerable effort needed to develop a register allocator is reduced to merely providing the architecture specific support modules to be coupled with a generic register allocator.

- Different register allocation strategies can be developed without tying them to a specific architecture, allowing for engineers to mix-and-match register allocators and backends to find the best combination for the particular performance goals sought.

- The register allocation modules in such a framework will be subject to much testing, since they will be used in conjunction with many different compiler backends. This decreases the chance that the register allocator will be the source of bugs in the compiler, making it a more highly trusted component in the compiler.

There have been other systems developed with generic register allocators. Sites' pseudo-code compiler[15] and Register Transfer Language system by McConnell et al.[7, part III] both demonstrated the advantages of developing a single register allocation module for use by all of their backends. However, the methodology used in those compilers was to define a low-level intermediate language with a precise semantics. Then, the system compiles the program down to that language, performs

register allocation on the low-level program (based on register information provided by backend-specific modules), and finally compiles the program into assembly code, propagating the register assignments through.

This approach works on simple RISC architectures, but using it to target CISC architectures is fraught with complications. The intermediate language could be defined as the least common denominator over all targeted architectures, in which case the low-level program to final assembly pass is either very complicated or produces poor code for CISC architectures. Alternatively, the intermediate language could support opcodes for complex instruction sets, but in that case some sort of constraint language must be developed to tell the compiler which opcodes in the intermediate language are actually valid for each architecture.

Additional constraints are placed upon the generated code by being shackled to a single intermediate language. Efficient architecture-specific code sequences may not be available if one chose "bad" register assignments in the allocation step, or if the intermediate language simply is not expressive enough. Also, the register set available to the allocator is hindered in the implementation described by Sites, since it partitions the registers into two sets: one for allocation and one for code generation. Such partitioning of the register set unduly limits allocation.

The framework presented here takes an entirely different approach. Rather than compile to a low-level language with precise semantics, allocate on that language, and then compile to the final assembly, this infrastructure allows the compiler to compile to the final assembly code and then do allocation directly upon it, but is able to use the same register allocation code base for any supported assembly language.

We perform the allocation in an architecture-blind fashion by defining an abstract *view* of certain properties maintained by the generated assembly code. The code generator is responsible for constructing the meta-information that this view is based upon, and the various compiler passes in turn are responsible for preserving the meta-information accordingly.

This approach is not without its limitations. Since the register allocator works on a view of the code, it does not have the precise semantics of each instruction

available for interpretation, rendering certain transformations impossible. In such a situation, it cannot perform transformations that inherently rely on inserting or duplicating arbitrary code at the assembly code level. However, this loss is justified by the usefulness of having a single interface joining the register allocator to the assembly code. If necessary, the generic allocator can be replaced by an architecture-dependent one that re-enables such optimizations; the generic allocator still would have served a useful purpose in getting the rest of the backend off the ground. Also, a generic allocator can be used as a source for trusted code to compare against when tracking down bugs in an experimental register allocator.

The rest of this thesis is organized as follows. Chapter 2 reviews the problem of register allocation as typically outlined in compiler texts, including a common technique of expressing allocation constraints using use/def information alone (a powerful idiom that this framework itself uses, where possible). It also lays out some of the dimensions available in the design space for register allocators, in order to bring home the point that there is no one best known technique for register allocation, and therefore defining an interface to allow plugging in alternate register allocators in different contexts is useful. It finally lays the foundation for the abstract view of the code exported by our assembly language representation. Chapter 3 discusses architectural register assignment constraints and the issues that arise when allocating programs with variables typed to hold particular kinds of values, and presents *joint-weighted graphs*, a graph abstraction that is well suited for generic register allocation. Chapter 4 presents the problems introduced by cross instruction register constraints, along with the solution of using *instruction groups* to abstractly describe such constraints. Chapter 5 describes the integration of the framework with the rest of the FLEX compiler infrastructure. Chapter 6 provides experimental results in using the infrastructure. Chapter 7 concludes and discusses potential future work.

Chapter 2

General Register Allocation

The general model used by modern processors is as follows: a working set of temporary variables are kept in the register file of the processor. When a computation references a variable that is not in the working set, the processor removes some variable from the register file (possibly writing its value to memory for later retrieval) and then evaluates an expression to generate the value of needed variable (possibly loading the value from memory into the register file), thus swapping one variable of the working set with another. At its core, register allocation consists of deciding what the working set will be at each point in the program, and inserting the necessary maintenance code, trying to minimize the number of memory instructions emitted [3, ch. 11] [12, ch. 16].

The problem of register assignment can be posed in the form of a resource allocation problem; variables consume registers, a finite resource. The essential question is this: at a given location in the control flow of the program, when it is safe to assign a particular register to some variable x? The answer: when no other variable is going to be loaded into that register while control can still flow to a spot that will refer to x.

Any body of code can be modelled as a directed graph of instructions, directly exposing control-flow in the program. This graph, coupled with information about what variables each instruction reads or writes, allows us to infer which variables interfere with each other. A variable x *interferes* with another variable y when x

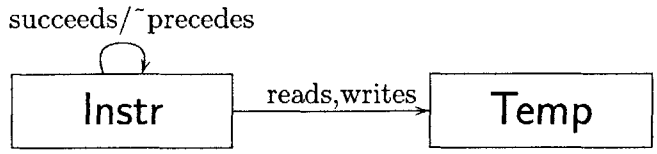

Figure 2-1: Simple Object Model of Computation

might be loaded into a register at a point where control could flow to a point where the value of y is needed before being reloaded. In this situation, it is clear that x and y cannot be allocated to the same register, because if they were, then the value of y would be lost when x was loaded into the register.

2.1 Alloy Model of Computation

In Figure 2.1 is a graphical depiction of the elements of the simple view of the problem and how they relate to one another; in subsequent sections this model will be elaborated with additional domains and relations, to clarify the wrinkles introduced by details described later.

The notation used is based on Alloy, an object modelling language developed by the Software Design Group at the MIT Laboratory for Computer Science [11]. Each box in the model represents a set of objects (called a *domain*). The arrows joining the boxes represent relations between the objects in the domains; a triangular arrowhead denotes subdomain relations. The notation "$A/\tilde{}B$" on a relation indicates that B is the inverse of A.

For this model, we have two domains. Instr is a piece of assembly code representing a single abstract instruction, such as "a $\leftarrow$ b + c" or "branch to L0 if a = 0". It may be realized by a (preferably short) sequence of assembly code instructions. Temp is a storage location holding some value to be used during the computation. In addition, we have four relations. The succeeds and precedes relations will be explained in section 2.1.1. The reads and writes relations will be explained in section 2.1.2.

2.1.1 Control Flow

We shall express the control flow of the program through the notion of a succeeds relation. An instruction i *succeeds* another instruction j if control might flow from i to j. From this relation we derive two useful auxilliary functions. The *predecessors* of an instruction i is the set of all instructions j_p where i succeeds j_p. The *successors* of an instruction is the set of all instructions j_s where j_s succeeds i.

One notion that the provided definition of succeeds does not explicitly model is that of control-flow exits. Instructions that conceptually represent the end of execution (such as returning from a function call or throwing an exception) are handled as a special case and are denoted as having no successors.

This convention handles commands like "`return;`" well, but we run into a pathological case where certain commands might have successors in the current code or might exit the procedure. Such commands cannot be modelled directly using this convention, because their set of successors would need to be simultaneously empty and non-empty. This case arises in Java code when modelling procedure calls with the possibility of throwing an uncaught exception; such calls may flow to the next instruction upon returning, or may throw the uncaught exception which then needs to be handled by a parent in the call stack.

We model such commands by as branching to all the successors in the code, plus an additional successor which itself has no successors. Thus if we had a hypothetical instruction where control could exit from the current function or could branch to a label, the appropriate way to model such an instruction would be as a branch with two successors, where one of the successors was the label and the other successor was a symbolic instruction with an empty set of successors. A common idiom is to have only one instruction representing the control-flow exit, and modelling all other exits as branching to the common exit.

2.1.2 Variable References

We shall relate instructions to the variables it references through the reads and writes relations. An Instr i *reads* a Temp x if the execution of i might require the value x held prior to entry to i. An Instr i *writes* a variable y if the execution of i might bind y to a new value. An instruction that reads x is known as a *use* of x, and likewise an instruction that writes x is known as a *def* of x.

Note that the writes relation does not necessarily denote the computation of a value that is actually intended for later usage, nor does it guarantee that the variable is written at all! In addition to modelling such situations, the writes relation also used to denote predicaments where a variable is *clobbered*; that is, where a variable's current value is lost but the value the variable currently holds is unspecified. However, if an Instr writes a variable, it must allow for the possibility of the register allocator assigning a new register to that definition.[1] An example of where this notion of clobbering is useful is in modelling caller-save registers; if we model a procedure call as a def of {`r0`, `r1`, `r2`, `r3`}, then the register allocator will know that values stored in those registers are not guaranteed to be preserved across the call, and will take the necessary steps to maintain values needed across the call in other registers or in memory.

It is the responsibiliy of the code generator to communicate the distinction between a true definition of a variable versus a clobbering of a variable; it does this by encoding it into the control flow of the code. In general, if a use of a register r is reachable from a def of r, without any intervening defs of r along that path, then the def is treated as a true definition which must be preserved. Otherwise the def is treated as a clobbering, and r can be safely overwritten through allocation to another variable.

Different languages (both high and low level) have varied conventions for indicating when variables are read from or written to. In order to keep things clear to those unfamiliar with a particular language, in the remaining chapters source code fragments will be annotated with marks to indicate which Temps in a given Instr are

[1] This is a result of allowing a generalized Instruction $\times$ Variable $\mapsto$ Register mapping. Section 4.2 further discusses the ramifications of this.

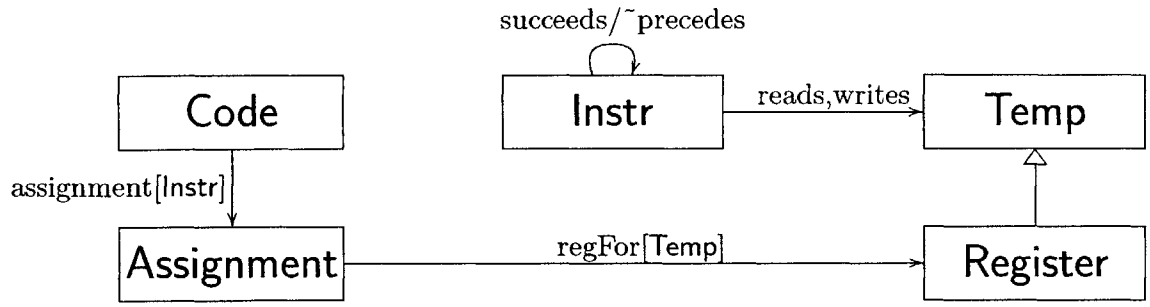

Figure 2-2: Object Model of Register Assignment

read or written. Temps which may be written will be marked with an underline, and Temps which may be read likewise with an overline. This convention was chosen as to minimize the possibility that the annotations could be mistaken as part of the original source code. As an example of these annotations, the Java statement "$\underline{z}$ = $\overline{x}$ + $\underline{\overline{y}}$++;" is an Instr that first reads x and y and then writes z and y.

2.2 Modelling Register Assignment

The model in Figure 2.1 presents an abstract description of the code for imperative language. We now extend the model with domains relevant to register allocation.

Figure 2.2 adds a domain to represent the finite set of registers for a given architecture, Register. Some Instrs require direct manipulation of particular registers. For example, the procedural calling convention used on the StrongARM requires passing arguments in specific registers. To model such Instrs, we make Register a subdomain of the Temp domain. This way specific registers can be included in the reads and defs relations for a given Instr.

The job of the register allocator is to discover a valid register assignment mapping an (Instr, Temp) pair to a Register. Thus we add a Assignment domain, which supplies a partial mapping from Temps to Registers, and a Code domain, mapping each Instr to an Assignment providing the registers for the Temps referenced by that Instr.

2.3 Graph Coloring Register Allocation

The insight of the graph-coloring interpretation of the problem is that the control-flow and the use/def information together yield an undirected graph of temporaries, where the edges in the graph represent interferences between variables[8].

This model of the problem is useful because it corresponds exactly to the problem of graph coloring. In graph coloring problems, no two adjacent nodes can be assigned the same color; analogously here, no two adjacent nodes can be assigned the same register. Thus we can apply graph coloring techniques to attempt to find a correct register assignment for a given program.

Finding if a k-coloring exists for an arbitrary graph is an NP complete problem. Chaitin recommends a heuristic for coloring the graph in which one pushes nodes with degree $< k$ onto a stack, removing each one (and its corresponding edges) from the interference graph as it is pushed. If one is lucky, the whole graph will eventually be pushed onto the stack, and thus one can then pop the nodes off the stack with the assurance that a color will be available for each one.

2.3.1 Spilling

Of course, a given graph may not be k-colorable. For register allocation, this corresponds to the situation where more variables are live at a point in the program than can possibly be loaded into the register file at once.

In general, there is no choice but to keep some variables in memory, storing them sometime after they are written and loading them into registers on demand before they are read. The process of putting aside space in memory and inserting maintenance code for such a variable is called *spilling*.

The design space for a spilling system is quite large. The most conservative approach to spilling is to choose some variable in the interference graph, allocate space in memory for the variable, and rewrite the instruction sequence with code to store its value immediately after each definition point, and reload the variable from memory immediately prior to each use. What variable to choose is important; Chaitin

recommends choosing a node v that minimizes the expression $cost(v)/degree(v)$ where

$$cost(v) = \sum_{\{I|v \in I.reads \cup I.writes\}} 10^{depth(I)}$$

and $depth(I)$ is the loop-nesting depth of I. Minimizing this expression reflects an attempt to minimize the negative impact of spill code on the program while maximizing the chance that the rewritten program will be colorable with the simplification algorithm.

The procedure then removes the node corresponding to the chosen variable from the original interference graph and continue coloring the graph. With each failed coloring attempt, the algorithm spills yet another variable to memory, adding new code for it and removing its node from the graph.

After the graph has been completely colored, however, the coloring process must actually be performed again, rebuilding the graph to account for the additional nodes corresponding to the new small live ranges made up by the spill code.

This approach effectively minimizes the size of each live range for the variable, replacing the original node in the interference graph into several new nodes (corresponding to the new live ranges) and hopefully yielding a k-colorable graph.

This is not the only approach to spilling available, though it is the one with the best chance of resulting in a k-colorable graph. Another approach to spilling a variable x is to only add a load before the first use of x in each basic block[2], and only add a store after the last def of x in each basic block [4]. All subsequent reads (or prior definitions) of x in the block are intended to be assigned to the same register for that block. This has the advantage of leading to less memory traffic when the generated code is executed. This approach to spilling can be implemented through a simple post pass to a conservative spiller that removes the intermediate spill instructions; such a post pass is commonly referred to as *cleaning*. Unfortunately, in general spilling with cleaning might never yield graphs that are k-colorable; it is entirely possible that the

[2]A *basic block* is a linear sequence of instructions, where control-flow only merges at the beginning of the block and only splits at the end of the block

live range *must* be split in the middle of a basic block. Therefore, cleaning must be used with care, to ensure that register allocation will actually succeed in the general case. The approach advocated by Bernstein et al. is to only apply cleaning on the first two coloring attempts for a given procedure, and then to use normal spilling thereafter.

2.3.2 Rewriting Code

In order to spill variables, we need to modify the assembly source code. This is more complicated than the simple mappings introduced to support register assignment. Register assignment is essentially putting values into slots that have already been set up and are waiting to be filled. Spilling requires actually inserting new code into the program. There are a number of strategies to handle dynamically adding spill code in an architecture independent register allocator.

If there is a mapping from the Temps used in the Instr form to the variables used in some prior intermediate representation, and if the code from that intermediate representation is saved during compilation, then one could rewrite the code in the intermediate form such that the variable references are replaced with references to a memory location. However, these conditions are not trivial to meet, and there is the possibility that the process of recompiling from the earlier representation will yield another uncolorable graph, ad infinitum.

An alternative idea is to simply have the backend provide a module responsible for generating spill code for a temp and for adding such code between any pair of Instrs joined by the succs and preds relations. This way at each Instr that references a Temp, we can insert the necessary code to load or store that Temp on the edges around it as needed. It is a little tricky to get such an implementation right though, as it needs to ensure not only that inserted code is executed when spilled Temps need to be stored or reloaded, but that the inserted code is *not* executed otherwise.

The approach used by the implementation described here, however, is a bit different. The Instr domain is extended with two new relations, prev_l and next_l. These relations denote *layout* in the instruction sequence, which may or may not correspond

to control-flow through the instruction sequence. Our backend does provides an architecture specific module, called an InstrBuilder, responsible for adding spill code, but instead of adding it between control flow edges, it adds it between layout edges. This complicates the register allocator slightly (since it now must determine where in the layout it should insert spill code), but simplifies the backend significantly.

As a constraint on the code generator, we do not allow branch targets to be uses.[3] In addition branching Instrs with computed targets are not allowed to be defs.

This approach is less abstract than inserting spill code at arbitrary control-flow edges, but it is simple for backend developers to understand and implement correctly, which was a design goal for this system.

2.3.3 Coalescing

A correct register allocator can be built with just the notions described above. However, such an allocator is quite limited in certain contexts and may produce significantly poor code. Notably, it ignores a particular optimization known as *Coalescing*[9]. Coalescing identifies variables that are copied from one Temp to another, but do not interfere with each other, and puts a higher priority of mapping such Temps to the same register, because then the redundant Instr "$\underline{\mathtt{r}_N} \leftarrow \overline{\mathtt{r}_N}$" can be eliminated trivially.

Coalescing is an extremely important optimization for the FLEX compiler because it uses SSA-based[4] forms for several of its intermediate representations. Compilers using SSA-form tend to produce numerous extra move instructions (also known as copy instructions) and will be doomed without an intelligent coalescing strategy.

Thus we extend our model to explicitly account for such instructions with the Move domain. As a special case of Instrs, Moves read a single Temp and write a single Temp, and each Move m must have the property that if the code were rewritten such that the sets m reads and writes map to the same Register, then m can be safely

[3]Since branch targets correspond to labels in the assembly output, this constraint is easy to satisfy.

[4]SSA stands for "static single-assignment," meaning every variable has only one definition in the program text[3, ch. 19].

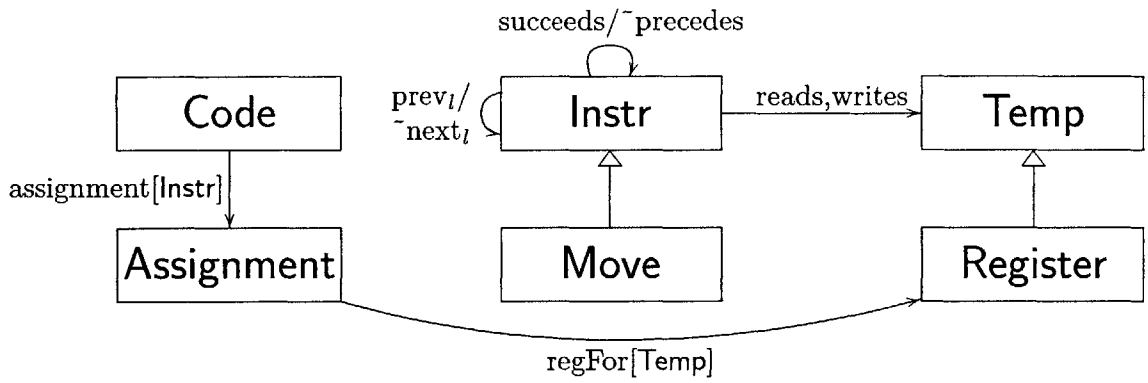

Figure 2-3: Object Model of Graph Coloring Register Allocation

eliminated from the instruction sequence. If either of these properties is violated, then m can no longer qualify as a Move.

The final model of Graph Coloring Register Allocation is given in Figure 2.3.3.

2.4 Related Work

Optimistic coloring was introduced by Briggs et al. as a way to significantly reduce spill instructions[14]. In essence, in this technique when a Temp is removed from the interference graph, it is not treated as a definite spill but rather a *potential* one. It is placed on the select stack like the other nodes. The subsequent attempt to color often can find a free color for the potential spill, simply because two neighbors of the Temp happened to be assigned the same colors (remember, Chaitin's $< K$ rule is conservative; if true, it ensures that a coloring will exist, but if false, a coloring may still exist). This technique is also recommend by Appel[3].

Another extension to the algorithm is to incorporate *multiple spill heuristics*[4]. The heuristic Chaitin gave for choosing a node out of the spill candidates is one of the best known. But like any heuristic, it can sometimes fail. Berstein et al. came up with three other heuristics to apply; they would build the graph once and checkpoint it. Then they would simplify the graph with each heuristic in turn, tracking the cost of the inserted spill code for each heuristic. Finally, they would choose the heuristic

that was associated with the lowest spill cost and commit to using that heuristic for the final graph coloring. This is called the "best-of-three approach."

One important point that has not been noted in the literature is that if one uses both the best-of-three approach and optimistic coloring simultaneously, it is not sufficient to commit to a heuristic before attempting to color the graph. The expected cost measured by the spill heuristic is based on the maximum number of the spills that could be inserted for that particular simplification; but with optimistic coloring, the actual number of spills is often less than the maximum. Moreover, there is little correlation between the spill cost predicted by each heuristic and the actual spill cost that occurs after optimistic coloring. This means that if the procedure commits to a particular heuristic before coloring the graph, then it could yield inferior code than if it had delayed commiting to a heuristic until *after* optimistic coloring has been applied.

2.4.1 Other Generic Register Allocators

In "Machine-Independent Register Allocation," Sites describes a pass over a low-level intermediate language that inputs variable names (coupled with interference and usage frequency information) and maps the names to a set of storage locations [15].His framework focused on allowing for the description of complex storage hierarchies with many levels of speed; our system does not account for anything more complicated than two levels of speed: registers (fast) and memory (slow). His system also exposed properties like bit-width to the allocation system; our framework abstracts such low-level details away.

Bryant et al. outline their planned interface for generic register allocation, "GRAS" in a technical report[6]. The problems they identify in their work seem to be quite similar to the ones discussed in this thesis[5]. Unfortunately, no sign of further development of their system is evident.

[5]Even down to the choice of names; Generic Register Allocation was among the options considered here

2.5 Summary

Processors need to shuffle values to and from memory during their execution. It is the job of the register allocator to decide how to best assign registers to the variables in the source code, and where to insert spill code that copies register state to and from memory.

If any abstraction of the code is going to be useful for register allocation, it needs to incorporate a notion of: *control flow*, *variable reference*, and *code layout*. It is also a good idea, for the purposes of code optimization, to expose other notions to the allocator, such as *move instructions*.

Chapter 3

Handling Assignment Constraints and Multiword Temps

Modern architectures present many constraints on register allocation. Sometime Temps of a particular type can only be kept in specific register banks for efficient operation. Other times single Temps require multiple registers to be allocated to them, and the registers assigned to the Temps may be constrained in certain ways.

3.1 Register Types

Variables are often kept in seperate register banks on modern processors. This has the advantage of seperating the contention for the different functional units; floating point and integer instructions can be executed simulataneously, allowing for more instruction level parallelism. High level languages like Java take advantage of this distinction and specifically indicate whether each variable is of type `int` or `float`.

However, such a design does complicate register allocation slightly, since it means that none of the registers available are truly general purpose; instead, each Temp has a specific set of registers that it can be assigned to.

In order to communicate this architecture-specific constraint to the register allocator, our infrastructure requires the backend-supplied `RegFileInfo` module to provide the functions

- `allRegisters()` :: () $\rightarrow$ set of Register
- `illegal`(t) :: Temp $\rightarrow$ set of Register

These functions allows the allocator to scan the code and introduce interferences between Temps and any registers that they can never be assigned to. This has the obvious effect of ensuring that filtering out a number of invalid colorings, but also properly increases the degree of the nodes to reflect the limited area that is available to them.

3.2 Register Clusters

Another problem presented when compiling languages like Java to a 32-bit processor is supporting 64-bit types such as `long` or `double`. Variables belonging to these types require 64 bits for proper representation, but each general purpose register on a architecture such as the StrongARM holds only 32 bits. Thus such variables require two registers to be allocated to them.

One approach to this problem is to represent two-word variables with two seperate temporaries: one for the low 32-bits and one for the high 32-bits. The two temporaries could be given distinct types, such as "low-of-long" and "high-of-long." Thus, a high-level bitwise `AND` of two longs like "$\underline{\mathtt{x}}$ = $\overline{\mathtt{y}}$ & $\overline{\mathtt{z}}$;" would be compiled into low level code like

```
and t0, t2, t4
and t1, t3, t5
```

where `t0, t2` and `t4` would be of type "low-of-long", and likewise `t1, t3` and `t5` would be of type "high-of-long." The interference graph will properly generate edges between the conflicting nodes, and so the allocation will proceed seamlessly.

Such an approach works transparently on architectures like the StrongARM, where longs and doubles can be allocated to any two registers in the register file. But this approach, while applicable to an architecture like the StrongARM, will not work in general; many architectures impose constraints on where the components of a two-

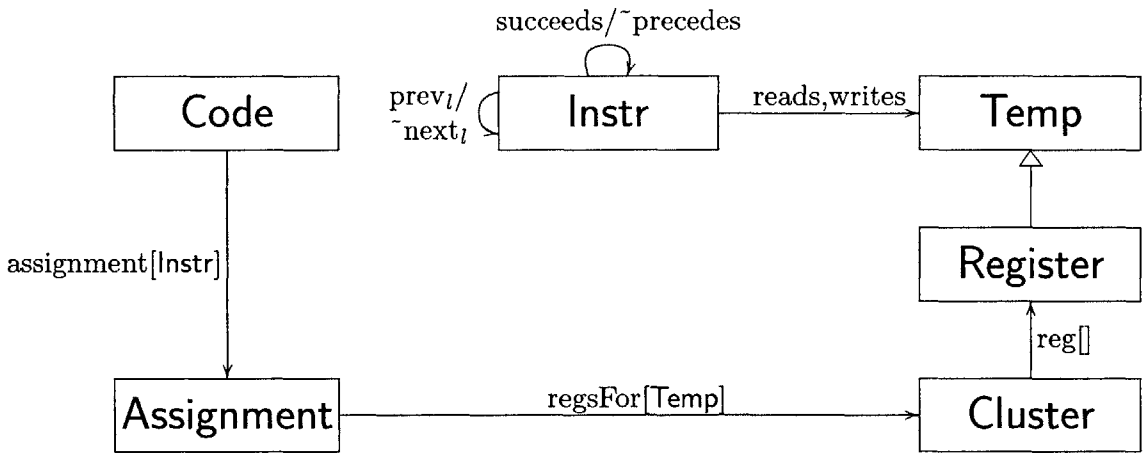

Figure 3-1: Object Model of Generic Register Allocation

word variable can go. For example, load-long instruction (`ldl`) on the Intel 80960 loads two words from memory into consecutive registers and requires that the index of the low order register be an even number[10, ch. 3]. Note that it is not enough to merely type the variables as being "low" or "high"; the relationship *between* the two components of the Temp.

As stated by Nickerson, "It would be easier for compiler writers if the multi-register operands were handled by widening the instructions so that they explicitly address all of the constituent registers. Then the existing register algorithms could be employed, placing the cluster consituents in whatever registers were avaiable in the region where the cluster is live. Both the added width of the instruction and the greater complexity of dereferencing the extra register addresses makes this suggestion unpalatable from a hardware design perspective." In short, the design of modern hardware imposes constraints on where longs and doubles can be allocated.

A truly general register allocator needs to account for these inter-register constraints. However, there is not much discussion of this issue in the literature [5] [13].

3.2.1 Problem Analysis

To properly reflect that the allocator now assigns register sequences rather than just registers, the object model needs to be updated with new domain, Cluster, which just represents a sequence of registers. The updated model is shown in Figure 3-1. No assumption is made within the allocator about properties of the registers within the sequence; perhaps the elements of the cluster are adjacent with the low register aligned on even boundaries, or perhaps they are not. The allocator is oblivious to these sorts of details.

To make this problem tractable, we shall assume that inter-register constraints only arise with respect to single variables being allocated to a sequence of registers. Thus, the backend module does not need to communicate inter-register constraints across multiple temps.[1] Different architectures place a variety constraints on what register clusters are legal assignments; we explore the design space in subsequent sections.

3.2.2 Nickerson's Approach to Register Clusters

Nickerson takes a novel approach to handling the constraints presented by Temps requiring register clusters [13]. His algorithm has each Temp map to a ordered set of nodes in the interference graph. Each set of nodes had a representative (the lowest register in the set). He would pass the graph through a normalization process where interference edges between non-representative members would be moved to the representatives of the respective sets, thus simplifying the graph and exposing hidden interferences between the Temps. He would then color the the registers accordingly.

Nickerson's approach, however, is strongly tied to one particular architecture (the Intel 80960 family). It presumes that each cluster is made up of an aligned continguous sequence of registers; this makes sense for that paper, where the targetted architecture infers register sequences from the base register and the number of registers involved.

[1] Cases of such constraints do arise in some situations; some techniques for handling them shall be covered in Chapter 4

However, an architecture independent allocator cannot make such assumptions.

Also, alignment constraints for his system are inferrable from the size of the register cluster. However, the inferred constraints may be wrong for other architectures; it may over-constrain some architectures, or may permit illegal assignments on others. So while Nickerson's approach is appropriate for a register allocator specialized for the Intel 80960, it cannot be applied in an architecture-independent manner.

3.2.3 Briggs' Approach to Register Clusters

Briggs took a different tact to handling register clusters [5]. Rather than always mapping each Temp to a set of nodes in the interference graph, he sometimes uses a multi-graph representation, where two nodes in the graph can have more than one edge joining them. The motivation for allowing multiple edges between a given node pair is that the extra edges enlarge the degrees of the nodes, which reflects the additional constraints presented by conflicts with Temps requiring register cluster assignments.

In addition to discussing architectural constraints on register assignment as a motivating factor, Briggs also notes another situation where aligning values into adjacent registers is useful. Some architectures have efficient instructions for loading several values from memory into adjacent registers; it is useful to take advantage of these instructions where possible by loading structures into adjacent registers from memory and storing them from adjacent registers into memory. While the structures are in the register file, they need not remain in adjacent registers; it is only during memory transfer that there is an advantage to keeping the value in adjacent registers. Briggs points out that a register allocator equipped to account for adjacency constraints is well suited such a situation because the compiler can output code to load the value into a register cluster and then output moves copying the components of the structure into other registers, and likewise output moves copying the value into another register cluster before output the store instruction. Thus the register allocator is free to decide if the structure can remain in the register cluster throughout its lifetime, or if it needs to assign the Temps in the moves to alternate registers in order to reduce

the complexity of the interference graph.

Briggs discusses three different types of graphs to build, each corresponding to different constraints in their respective target architectures.

- "Unconstrained Pairs" corresponds to when some Temps allocate some pair of registers drawn from the register file, and the assigned pairs need not be aligned or even adjacent. For this case, Briggs builds a graph with two nodes for each Temp, and represents the conflicts accordingly.

- "Adjacent Pairs" corresponds to when the target machine requires that pairs be allocated to aligned adjacent registers. Here Briggs suggests building a multigraph where both (double, single) and (double, double) node pairs have two edges between them.

- "Unaligned Pairs" corresponds to when the target machine requires that pairs be allocated to adjacent registers but the pairs need not be aligned. Surprisingly, removing this constraint on the available assignments actually introduces *more* constraints on the interference graph. Briggs handles this by building a multigraph where again (double,single) node pairs have two edges between them, but now (double,double) node pairs have *three* edges between them.

Briggs' work only dealt with assigning pairs of registers to Temps. This presents a problem for compilers wishing to have Temps that are assigned to more than two register slots, such as compilers which keep reasonably sized records in the register file rather than in memory. The only consideration given to the problem of allocating larger register clusters is restricted to the statement "our scheme extends in a straightforward way to larger aggregate register groupings." The actual extension is not specified; the infrastructure to be presented here might be considered a suitable candidate.

Also, requiring the construction of different multigraphs seems at first to introduce a depedency on the target architecture, which is the same reason that we dismissed Nickerson's approach. However, here we present a way to construct such graphs

in an architecture independent fashion, by moving the information yielding higher degrees into the backend-specific module. This way, the register allocator will scale the degrees of the nodes up according to how many constraints are introduced by interferences between them, without needing to actually know specific details about the specific alignment or adjacency constraints at the graph construction stage.

3.3 Joint-Weighted Graphs

To use Briggs' approach, we need to cope with adding more information to the edges between the nodes, and we need to extend the graph building routine to retrieve this information from the backend module.

To support this, we require the backend module to provide the following functions to the register allocator:

- `occupancy`(t) :: Temp $\rightarrow$ Natural
- `pressure`(t_a,t_b) :: Temp $\times$ Temp $\rightarrow$ Natural

occupancy(t) maps to the number of registers t will occupy in the register file when assigned. *pressure*(t_a,t_b) maps to the amount that t_b would raise the degree of t_a if t_b and t_a were found to interfere. An important detail is that *pressure*(t_a,t_b) is **not** necessarily symmetric!

Given proper implementations, these two functions guide the register allocator to construct the appropriate graph forms (handling all of the variants Briggs presented[2]), while the algorithm within the allocator itself remains architecture independent. The main difference in the register allocator is this: instead of having a multigraph with many edges joining the nodes, the register allocator now has a graph with at most one edge joining each pair of nodes, and each edge has *two* weights associated with it; one for each node that it is joined to!

Consider the following example illustrating this abstraction. Let `j` and `k` be Temps requiring assignments to register pairs, while `i` will occupy a single register once

[2]and possibly others, to deal with even more exotic architectures

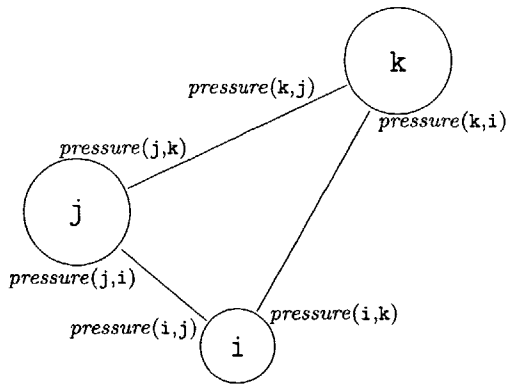

Figure 3-2: Example of Joint-Weighted Graph

assigned. Furthermore, assume that all three of these Temps interfere with each other.

Using the *occupancy* and *pressure* functions, the generic register allocator builds the graph presented in Figure 3.3. The degree of each node is calculated by summing the weights contributed by each of the edges joined to it. For example,

$$degree(\mathtt{i}) = pressure(\mathtt{i}, \mathtt{j}) + pressure(\mathtt{i}, \mathtt{k})$$

For all of the following cases, the *occupancy* function is

$$occupancy(t) = \begin{cases} 2 & \text{if } t \text{ requires a register pair assignment} \\ 1 & \text{otherwise} \end{cases}$$

Thus, $occupancy(\mathtt{i}) = 1$, $occupancy(\mathtt{j}) = 2$, and $occupancy(\mathtt{k}) = 2$.

The cases of "Adjacent Aligned Pairs" and "Adjancent Unaligned Pairs" are the most straightforward, because *pressure* is symmetric for these two cases. The case for unconstrained pairs is more interesting, because the use of an asymmetric *pressure* function allows the allocator to properly use the fact that in such architectures, high occupancy Temps have a much larger impact on the degrees of low occupancy ones than vice versa.

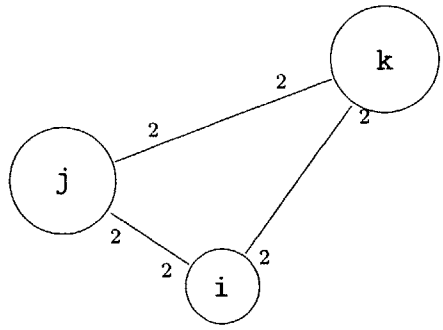

Figure 3-3: Joint-Weighted Graph, Adjacent Aligned Pairs

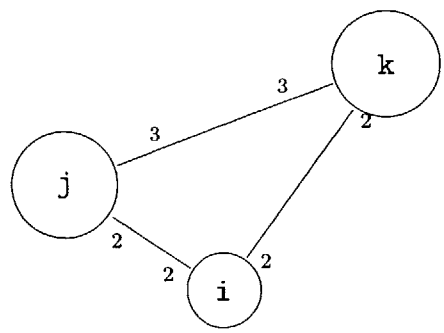

Figure 3-4: Joint-Weighted Graph, Adjacent Unaligned Pairs

Adjacent Aligned Pairs

For architectures requiring pairs be aligned, the appropriate *pressure* function is:

$$pressure(a, b) = \begin{cases} 2 & \text{if } occupancy(a) = 2 \text{ or } occupancy(b) = 2 \\ 1 & \text{otherwise} \end{cases}$$

As Figure 3.3 shows, this definition for *pressure* causes each edge in the diagram to increase the degree of the node it joins to by 2, which is exactly how much the degree would increase by when using Briggs' multigraph rule for adjacent aligned pairs.

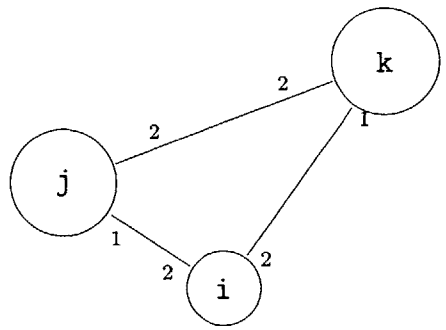

Figure 3-5: Joint-Weighted Graph, Unconstrained Pairs

Adjacent Unaligned Pairs

For architectures only requiring pairs be adjacent , the appropriate *pressure* function is:

$$pressure(a,b) = \begin{cases} 3 & \text{if } occupancy(a) = 2 \text{ and } occupancy(b) = 2 \\ 1 & \text{if } occupancy(a) = 1 \text{ and } occupancy(b) = 1 \\ 2 & \text{otherwise} \end{cases}$$

As Figure 3.3 shows, this definition for *pressure* causes the edges joining mixed occupancy nodes to have weights of 2, while the edge joining `j` and `k` increases their respective degrees by 3. Again, this is exactly how much the degree would increase by if one were to use Briggs' multigraph rule for adjacent unaligned pairs.

Unconstrained Pairs

For architectures where there are no constraints placed upon pairs chosen for a register assignment, but where the computer must keep both pairs alive simultaneously, the appropriate *pressure* function is:

$$pressure(a,b) = \begin{cases} 2 & \text{if } occupancy(b) = 2 \\ 1 & \text{otherwise} \end{cases}$$

This is fairly intuitive, though after working with the Briggs' solution for a long time, it takes a significant amount of thought to be convinced of its correctness.

When a high occupancy Temp like j interferes with a Temp like i, it is going to take two registers away from the selection available for i's use during color assignment. However, i is only going take one register from j's selection, **unlike** the case for the "Adjacent Aligned Pairs" where it took two registers away, because now the allocator is not forced to keep the pair of registers adjacent. For example, if there are three registers available, i can take any one of them, and j can still be successfully assigned to the other two.

Finally, for architectures where there are no constraints at all on the pair assignments, and where the live ranges of the pair components need not match, the most appropiate approach is to generate a different Temp for each pair component (and use the simple function $pressure(a, b) = 1$), since that best reflects the lack of constraint presented by the architecture.

3.3.1 An Unexpected Benefit of Pressure

While all of the uses of *pressure* so far have only increased the degrees of the nodes in the interference graph (thus making the problem of register allocation harder in general), there is one place where the function actually makes allocation easier. Temps that are due to be assigned to seperate register banks can never constrain eachother's coloring. Therefore the *pressure* function can return 0 for such pairs of Temps.

Thus, the compiler can perform register allocation for all of its Temps at once, without fear of some nodes in the interference graph being over-constrained due to conflicts with nodes that have no impact on the final coloring.

3.4 Mapping Temps to Assignments

After the interference graph is built, it needs to be colored. The standard coloring techniques can be applied, but because the allocator does not have a view of the Temps that exposes the underlying registers such algorithms expect, the allocator cannot actually perform a direct color to assignment mapping. So once again we need to appeal to the module provided by the backend.

However, this extension is much simpler than the *occupancy/pressure*function pairing given earlier.

- `assignment`(t,*occupied*) :: Temp $\times$ set of Register $\rightarrow$ Cluster

During the coloring process, the register allocator tracks the registers in use. Therefore, on each Temp t it can call out to the RegFileInfo, passing in the registers that have already been assigned, and get back an appropriate sequence of Registers for t. The case where no appropriate sequence is available is treated as equivalent to the case in the standard algorithm where no more colors are available: t is marked to be spilled before the next attempt to allocate.

3.5 Summary

Modern architectures often constrain register allocation with respect to assigning registers to large values. The edges between the nodes of an interference graph need to be updated with additional information to properly reflect these constraints. In order to seperate this additional information from the register allocation, a *joint-weighted graph* is a general abstraction that is flexible and easy to understand.

The interface of the backend hooked into the register allocator needs to provide functions that the allocator can call to obtain information relevant to building this graph and to properly assign registers to variables. The interface is summarized in Appendix A.

Chapter 4

Handling Cross Instruction Assignment Constraints

Many expressions in an intermediate representation for a compiler have a seemingly simple semantics but can require many assembly instructions to be realized properly. An example of this is the Java statement "`boolean` $\underline{\mathtt{b}}$ = $\overline{\mathtt{x}} > \overline{\mathtt{y}}$;" which compares the value of the variable `x` to that of the variable `y`, storing a value representing `true` into the variable `b` if `x` is greater than `y`, `false` otherwise.[1]

Conceptually, this sounds like something that should map directly to a single instruction that takes a destination register and two source registers as arguments, comparing the values in the sources and storing the result in the destination. On many architectures, this is precisely how such an operation is encoded.

4.1 Conditional Execution on the StrongARM

However, some architectures cannot take such a simple approach to properly implement the statement "`boolean` $\underline{\mathtt{b}}$ = $\overline{\mathtt{x}} > \overline{\mathtt{y}}$;" StrongARM, for example, does not have a less-than instruction taking three register operands. Instead, the StrongARM archi-

[1]Let us assume that `x` and `y` are both single word integer variables, and that the boolean values `true` and `false` are represented by the integers `1` and `0` respectively

tecture has four special single-bit flag registers, {N, Z, C, V},[2] and an all purpose cmp instruction that takes two register opferands.

When the cmp instruction is executed, the processor sets all four flags based on the difference of the two arguments.

These flag bits establish the basis for the StrongARM's notion of conditional execution. "All ARM instructions can be conditionally executed, which means that their execution may or may not take place depending on the values of the N, Z, C, V flags ..." [1, ch. 3]. Conditional execution is notated in the assembly language by a suffix appended on to the opcode for a given instruction. Conditional execution is a useful architectural trick, because it allows for many common instruction sequences to be expressed concisely in the assembly code.

The StrongARM complicates compilation of languages where booleans are first-class values that can be passed as arguments, moved from one variable to another, or stored into fields in memory. The StrongARM inherently makes boolean comparisons a special class of operations that do not work in the same manner as the other logical instructions, leading to a need to compile boolean operations like less-than in special ways.

4.2 Expansion into Multiple Instructions

To properly support the semantics of the statement "boolean $\underline{b}$ = $\overline{x} > \overline{y}$;" on the StrongARM architecture, we need to expand it into three assembly instructions:[3]

```
cmp   x̄, ȳ        @ compare x̄ to ȳ
movgt b̲, #1       @ if x > y , b̲ := true
movle b̲, #0       @ if x ≤ y , b̲ := false
```

This code properly expresses the desired semantics. However, it hints at a problem for our chosen abstract view of the assembly. Generating one Instr for each assembly

[2]the meaning of the different flags depends on the assembly instruction that set them, but in general N indicates "negative value," Z indicates "equal-value" (aka zero-difference), C means "carry-bit set," and V means "overflow"

[3]movgt and movle are conditionally executed variants of the mov instruction, which simply copies a value into a register. The @ is the comment character for StrongARM.

instruction in this code means that both the `movgt` and the `movlt` instructions are treated as writes of `b` by the abstraction of the assembly language that the register allocator sees.

However, the problem is that the allocator interprets this situation as meaning that `b` will be clobbered once and then written again. Therefore, an allocator might reason, it is safe to assign `b` to one register in the `movgt` command, and assign `b` to a *different* register in the `movle` command! For example, the allocator might produce code that looks like this:

```
cmp   r2̄, r3̄        @ compare x̄ to ȳ
movgt r0̲, #1        @ if x > y , b̲ := true
movle r1̲, #0        @ if x ≤ y , b̲ := false
```

Subsequent code would treat `r1` as the container of `b`. But if `x` > `y`, then `r0` actually holds the intended value of for `b`, and `r1` is not `b`, but instead some unknown value from some previous computation in the code!

This is a serious problem, and it complicates nearly every usage of conditional execution. It is also a problem with a wide range of solutions.

4.3 Assembly String Coalescing

One of the simplest solutions is to *coalesce* the seperate instructions into a single Instr. This can be done by using a multiple line assembly string when constructing the Instr. For example, the above assembly sequence, which was formerly represented by a series of three Instrs, is now one Instr:

```
cmp x̄,ȳ \n movgt b̲, #1 \n movle b̲, #0
```

Instead of three Instrs (a read of x and y followed by a pair of writes of b), we have a *single* Instr that reads x and y and writes b. In the actual executed assembly, this ends up being three instructions, and so use of this technique implies that each Instr does not necessarily correspond to a single assembly instruction. This could be a potential problem if pieces of the compiler use number of Instrs as a metric for estimating code size.

There is another obvious problem with assembly string coalescing as well. The technique hides the underlying logic from the other architecture-independent portions of the compiler; this could preclude other optimizations that would require such logic to be exposed. Also, other architecture-specific modules in the compiler backend might prefer to see the atomic instructions as Instrs. In essence, such an approach could be crippling[4] the whole compiler backend for the sake of the register allocator.

We naturally arrive at the question, "is this the correct approach for a compiler infrastructure to take?" Let us temporarily ignore the hypothetical repercussions that assembly string coalescing could have on the other portions of the compiler, and refocus on how the approach impacts on just the register allocation.

4.4 Cross Instruction Constraints

As it turns out, assembly string coalescing alone is not enough to create a code generator that can be used with a generic register allocator. An example of this comes up on the StrongARM.

The StrongARM architecture has a multiply operation, `mul`, that takes three register operands: a destination and two source registers. A curious aspect of the architecture, however, is that it requires that the destination register and the first source register be *distinct*. That is, the assembly instruction "`mul r0, r0, r1`" is illegal on StrongARM, while "`mul r0, r1, r0`" is legal.

One could communicate such constraints to the allocator by extending the Instr interface with some sort of querying system that the register allocator would have to verify potential combinations of assignments. Such an extension seems unnatural, however. It is not clear how to incorporate such queries into the register allocation without requiring some form of backtracking in the graph coloring or switching to a more general constraint solving system.

[4]Perhaps "crippling" is too strong a word, since one might argue that other compiler passes could actually parse the assembly string and infer the information it needs from that, or the information about the components of an Instr could be generated and associated with it in some other manner in the code-generator. Nonetheless, requiring such hacks in the other portions of the compiler is a blemish on the design of the system.

There is an easy way to encode such a constraint into the code itself without needing to add significant extensions to our existing system. In the Instr "`mul x, y, z`," the only way that `x` and `y` could be mapped to the same register would be if they did not conflict; that is, if this Instr was the last use of `y`. While there is no easy way to predict if the code-generation will yield a succeeding use of `y`, we do not need to make such predictions; we can just insert one ourselves! By emitting a *fake* use of `y` succeeding the multiply Instr, we can keep `y` alive, regardless of whether it is actually used in the succeeding code or not. We have coined the term *Dummys* for such fake uses of a Temp. Thus, the final assembly sequence for the multiply operation is:

```
mul x, y, z
@ dummy use of y
```

It is not only for cases like the StrongARM `mul` instruction that Dummys are useful. They come up most often to handle cross instruction constraints for operations involving multi-word Temps. For example, a move from `y` to `x`, where `x` and `y` are both two word temps, is implemented as follows on the StrongARM:

```
mov xl, yl
mov xh, yh
@ dummy use of yh
```

The reason for the Dummy is to ensure that `xl` is not assigned to the same register as `yh`, which would cause the data to be overwritten before it is able to be copied to `xh`.

While this would not be a concern for an architecture which required that the low-word of double-word Temps be assigned to even registers and the odd-word to be assigned to odd registers (since `xl` and `yh` could never be assigned to the same register in such a situation), the StrongARM architecture itself does not have such constraints. Almost all of the assembly sequences involving multi-word Temps need Dummys to rule out illegal register allocations.

4.5 Dummy Instrs and Spilling

While in principle Dummys seem like a transparent fix to the problem of communicating odd register constraints to the register allocator, there is a subtle problem with them. The reason for introducing Dummys was to keep particular Temps alive past a certain point. But the purpose behind inserting spill code is to *break up* live ranges. It is not hard to predict that the two will interact in contrary ways.

A concrete example of this can be illustrated with the multiply operation introduced earlier. Here is the assembly sequence for the operation:

```
mul x, ȳ, z̄
@ dummy use of ȳ
```

Now, imagine if y was defined much earlier in the code, and thus it interferes with many other Temps in this method. If the allocator spills without cleaning (because, for example, it has already gone through two coloring attempts with cleaning), and decides to spill y, then the assembly code will be changed as follows:

```
spill load of y
mul x, ȳ, z̄
spill load of y
@ dummy use of ȳ
```

This defeats the whole purpose of having Dummys. The second spill load acts as a write of y; this means that y will no longer be live across the `mul` instruction, and so the register allocator could allocate it to the same register as `x`.

This reveals a serious contention between two of the channels through which the code-generator communicates constraints to the register allocator. To stop spills from being inserted at key points, we coalesced instructions into a single Instr. But if we coalesce a Dummy into its predecessor, we will be back where we started, because a coalesced Dummy will no longer be visible to the liveness analysis (indeed, to coalesce a Dummy is to effectively remove it). So we have a concrete spilling problem which assembly string coalescing alone cannot solve.

4.6 Beyond Assembly String Coalescing

The basic problem now is that the spiller has too much freedom in terms of where it can insert spill code. To continue using the trick of keeping variables alive to communicate allocation constraints, it is necessary to tell the spiller where it is not allowed to spill.

One response to this problem is to have the spiller give special treatment to the Dummys. If we somehow expose the Dummys to the spiller, then it can deliberately **not** insert spill code before Dummys. Given such a spiller, spilling y in our above example will yield the following:

```
spill load of y
mul x, y, z
@ dummy use of y
```

This is exactly the effect desired. The live range of y has been split, thus reducing its impact on the interference graph, while the constraints communicated by keeping y alive across the `mul` Instr are maintained even after the spill code is inserted.

However, giving Dummys special treatment by the spiller requires that the spiller can identify which Instrs are Dummys. While a simple pattern match for a prefix like "`@ dummy`" in the assembly string might suffice in an architecture-dependent spiller, it will not be so here. Thus to use this approach, the infrastructure needs an explicit tag on the Instr type like `isDummy`. This is a little ugly.

Another problem is that there may be long sequence of assembly instructions where spills are not allowed to be inserted anywhere in the sequence. For example, a real-time system needs to provide hard guarantees of when certain operations will terminate, and inserting spill code might break those guarantees. To ensure such guarantees using this system, one would need to explicitly tag every instruction in the sequence as a Dummy.[5] Note that it would not be sufficient to just interpolate Dummys with the sequence of real instructions, since the spiller would be forced to

[5]Obviously in this situation the "`@ spill`" assembly string prefix will not suffice as a tag, since we want to tag real instructions. This notion of tagging so-called "real instructions" as Dummys reeks of confusion at best and contradiction at worst.

skip over the Dummys but could then insert spill code before the other Instrs, thus breaking up the sequence and invalidating the atomicity guarantee.

Finally, we again ask the question, "is this the correct approach for a compiler infrastructure to take?" Dummys and coalesced assembly strings seem to be at odds with each other, so perhaps the right approach is to step back and evaluate other options to solving the problem of communicating register allocation constraints from the code generator.

4.7 InstrGroups

At the heart of this problem is the idea that the compiler generates sequences of Instrs that we wish to treat as atomic units during register allocation. Assembly string coalescing simply turns the sequences of Instrs into a single Instr. Dummys are just Instrs that have been added to the end of a sequence that are given special treatment by the allocator.

A better option than trying to approximate these sequences with such hacks is to make the sequences explicit in the compiler infrastructure. One approach considered for representing such sequences was to introduce a CompositeInstr type that was an Instr and yet itself was composed of Instrs. This is a tempting prospect since it sounds like it would allow for seamless integration with existing code using Instrs. However, this approach was discarded as needlessly complex; finding a useful definition for the semantics of how the CompositeInstr would relate to its components is a difficult problem, and just one extra level of description above the Instrs is all that is required, not the generality provided by a *recursive* structure like CompositeInstr. The final resolution was to use InstrGroups.

An *InstrGroup* is a collection of Instrs that can be treated as a semi-atomic unit, or can have its underlying components exposed. Since different components of the compiler may want to see different levels of abstraction of the underlying Instrs, each InstrGroup has a single GroupTypeassociated with it. Thus each compiler pass may ask for a view of the code of a certain GroupType, and the system will provide a view

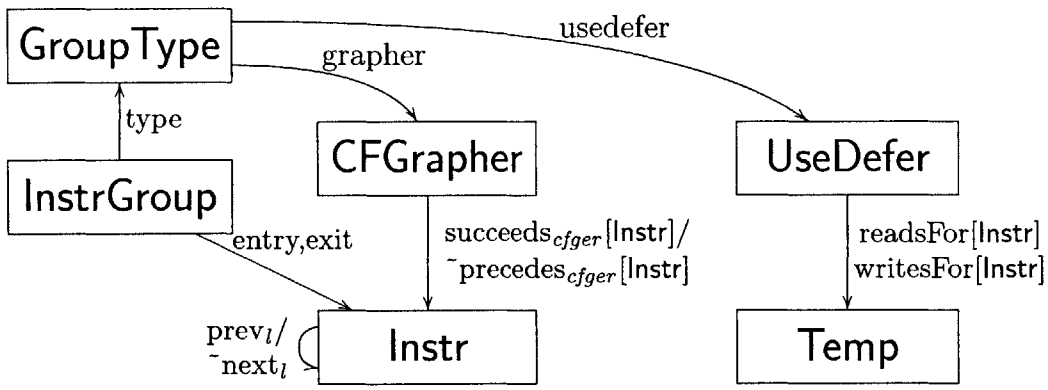

Figure 4-1: Object Model of Generic Register Allocation

which does not expose the components for groups of that GroupType. If a pass does not ask for a view of a specific GroupType, the Instrs are exposed as if there were no InstrGroup annotations (the lowest level view).

The *views* that are associated with the GroupTypes for the InstrGroups take the form of two objects, a UseDefer and a CFGrapher. These two object simply factor the reads/writes (from section 2.1.2) and succeeds/precedes relations (from section 2.1.1) out of the Instr object, and give them first class status. Now the allocator can choose whether it wants to see a view of the code that reveals where spilling is allowed, or a view that abstracts away the components of aggregate instructions within the generated code.

Intuitively one might think that the abstract external view of an InstrGroup should be a single Instr. That will not achieve much more than coalescing the assembly strings, however. Instead, we expose **both** the entry and the exit to the outside world.

The collection of Instrs within an InstrGroup must form a *single-entry/single-exit* region: that is, there must be one *entry* Instr that precedes any of the other Instrs in the group during any execution, and likewise a *exit* Instr that succeeds any of the other Instrs in the group during any execution. This allows for a natural abstract view of the control flow for the collection of Instrs within an InstrGroup g as just the

pair of Instrs $\langle g.\text{entry}, g.\text{exit}\rangle$, where g.exit is viewed as the sole successor of g.entry and vice-versa.

Since the register allocator is not supposed to look into the internal workings of abstract aggregate instructions, we are forced to take a conservative stance on them. For a given aggregate a, if any Instr in a writes a Temp t, then the register allocator will see a.entry as a def of t. If any Instr in a apart from a.entry reads a Temp t', then the register allocator will see a.exit as a use of t' (The Temps a.entryreads can safely remain associated with a.entry).

The reason for this strange mapping is to ensure that an illegal assignment is not performed if the backend writer is careless and reused the same Temp in many different ways within the aggregate grouping; by associating the Temps written with the entry and the Temps read with the exit, the allocator will maximize the number of interferences. However, this is not something to be proud of. It greatly constrains allocation; perhaps other group types could be introduced that the register allocator would take a less conservative action with.

4.8 Summary

Various instruction patterns produced by a code generator sometimes have implicit constraints that the register assignment chosen by the allocator must not break. To communicate such constraints, it is useful to define some way of grouping Instrs together, so that the constraints will be associated with them properly.

Two such groupings that we have found a need for are *aggregate* and *no spill* groupings. The former is used to hide away variable references that could confuse the allocation, and the latter is used to keep the allocator from inserting spill code in places that could break or hide implicit constraints.

Chapter 5

Implementation within the Flex Compiler Infrastructure

This infrastructure for architecture independent register allocation was designed and developed concurrently with the rest of the backend for the FLEX Compiler Infrastructure. The FLEX system compiles Java bytecodes as a source language to a wide variety of targets, from Java bytecode to C to low-level assembly code.

5.1 Modules of the Flex Compiler

As illustrated in Figure 5.1, the Flex compiler passes code through six distinct intermediate representations. Starting from Java Bytecodes, the compiler constructs a simple high-level operational code called QuadSSI, an extension to SSA-form de-

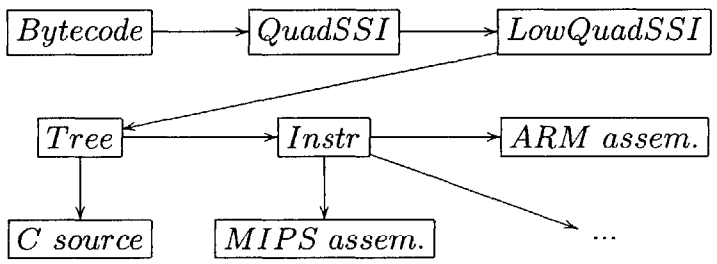

Figure 5-1: Phases of Flex Compilation

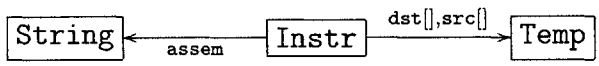

Figure 5-2: Class Definition for Instr type

scribed in Ananian[2].

After a series of optimizations on this representation of the code, it is then converted into a lower-level form named LowQuadSSI, which preserves the SSI properties but exposes details such as array indexing operations. This is then converted into an expression tree form, appropriately named Tree form.

The compiler can then convert the tree expressions directly into C source code, or it can use a tree-pattern-matching code-generation pass to convert the Tree form into the architecture specific Instr form. Each backend targetting Instr form provides a backend specification file providing a mapping from tree-patterns to Java code sequences for emitting the appropriate series of Instrs. The majority of the work of developing a backend for a specific processor is spent developing these specification files; it is there that the Instrs are constructed and that the relationships between Instrs, Temps, and InstrGroups are established.

5.2 Instr form

In the backends of the FLEX infrastructure, the basic units are Instrs and Temps.

As shown in Figure 5-2, each Instr keeps an immutable string of characters corresponding to its assembly code, and two arrays of Temps corresponding to the Temps it writes and reads. Since the assembly string is assigned at each Instr's creation, it obviously does not have references to the assigned registers directly written in the string. Instead, each string is a *template* for the final assembly code. The template has escape sequences embedded in the string marking where real register references will be placed during the final code output. Each escape sequence indicates which Temp is actually referenced at that point, so that when the register allocator provides the Temp $\rightarrow$ Register mapping, the appropriate register will be inserted in place of

the escape sequence.

For example, an Instr with `assem` = `"add ‘d0, ‘s0, ‘s1"` , `dst` = { `t0` } , and `src` = { `t1, t0` } , coupled with an Assignment mapping { t0 → r3, t1 → r6 }, will map to the assembly string `"add r3, r6, r3"`.

To handle Temps that map to register clusters, escape sequences may have backend-defined suffixes. These suffixes are used by the backend to decide which register in the cluster should be substituted for the Instr. On the StrongARM, the suffixes used were ‘l’ and ‘h,’ standing for “low” and “high” word, respectively. Thus the StrongARM assembly string sequence for adding two long variables is: [1]

```
adds ‘d0l, ‘s0l, ‘s1l
adc  ‘d0h, ‘s0h, ‘s1h
@ dummy use of ‘s0l ‘s0h ‘s1l ‘s1h
```

5.3 InstrGroups

In addition to maintaining an assembly string and collections of referenced Temps, each Instr also maintains a reference to the InstrGroups it is a member of. Currently, our system imposes the restriction that all InstrGroups be properly nested; this was done to ease the conceptual complexity of InstrGroups. This restriction, however, also allows for a simple representation for the Instr to InstrGroup mapping. Each Instr maps to a single InstrGroup, which in turn maps to either `null` or its parent InstrGroup. This forms a natural tree structure for each collection of nested groups (though it seems so far that there has been little need for branching in the generated trees).

5.4 Code Generation

The code generation module in the infrastructure consists of a specialized tree-pattern matcher which traverses the tree intermediate representation and emits instructions

[1]The reason for the adds and adc opcodes is to allow for a carry-out from the first addition to be incorporated into the second; the adds operation sets the condition codes (one of which maintains a carry bit) and the adc uses the carry bit in its addition.

```
BINOP<l>(ADD, j, k) = i %{
    { InstrGroup nsg = startGroup(InstrGroup.NO_SPILL);
    { InstrGroup agg = startGroup(InstrGroup.AGGREGATE);
    emit( ROOT, "adds `d0l, `s0l, `s1l", i, j, k );
    emit( ROOT, "adc  `d0h, `s0h, `s1h", i, j, k );
    endGroup(agg); }
    endGroup(nsg); }
}%
```

Figure 5-3: Sample Pattern from specification file (long addition)

according to the expressions that it matches. In order to seperate the patterns from the matching algorithm, the system has specially defined `CodeGen.spec` files which map tree patterns to the Java code for emitting an instruction sequence. Figure 5.3 shows a sample pattern from this file which generates the instructions for long addition.

The `CodeGen.spec` files are transformed into proper Java source code by a `CodeGeneratorGenerator` which outputs an appropriate tree pattern matching system, but specialized around the patterns defined in the file. Each architecture has its own `CodeGen.spec` file containing the particular pattern to code mapping.

5.5 Register Allocation

After assembly code is generated for a procedure, the code is passed through the generic register allocation module, along with a reference to the appropriate backend module (see Appendix A). The register allocation module uses the views of the code provided by the InstrGroups for its analysis, modifies the instruction sequence with spill instructions if necessary, and finally provides the code with an Instr $\rightarrow$ Assignment mapping.

Chapter 6

Experimental Results

To evaluate the performance impact implied by use of this register allocation infrastructure, two register allocators were constructed. One allocator specifically targeted the StrongARM processor architecture; the other used the interfaces defined here for architecture independent allocation.

To compare the two allocators' performance on a reasonable range of programs, a selection from the SPEC JVM98 benchmark suite was compiled using the FLEX compiler infrastructure. Both allocators used the same code generators to construct their inputs. The post-allocation assembly code was then instrumented to obtain dynamic instruction execution counts. Instrumentation code was also inserted to identify three subdomains of interest: move instructions, memory instructions, and spill instructions. Spill instructions generated by the system were annotated with special comments, so the instrumention was able to distinguish general memory instructions from spill instructions.

Dynamic execution counts for these domains are presented below. Raw timing data for the benchmark suite is also presented, as well as compilation times. All of the execution benchmarks were executed on a StrongARM 110 processor with 128 megabytes of memory, running the Linux operating system. The compilation times were gathered on a 1 Ghz AMD Athlon with 512 megabytes of memory, running on Suns 1.3 Java Development Kit on the Linux operating system. The raw times were gathered by running each benchmark five times, discarding the lowest and highest

Spec Benchmark	All Instructions	Move Instructions	Memory Instructions	Spill Instructions
_200_check	5,499,511	981,208	983,605	7,761
_201_compress	17,395,576,419	1,644,740,501	4,274,134,863	6,995
_202_jess	(linkage failure)	604,207,522	928,194,164	32,386,818
_205_raytrace	5,034,705,929	948,528,039	1,212,376,854	100,001,342
_209_db	8,870,965,382	2,105,328,042	1,664,124,468	392,777
_213_javac	4,410,618,130	927,303,310	946,620,985	21,565,187
_228_jack	5,344,361,324	578,685,249	1,507,589,986	38,802,559

Figure 6-1: Dynamic Execution Counts, Generic Register Allocation

Spec Benchmark	All Instructions	Move Instructions	Memory Instructions	Spill Instructions
_200_check	5,660,219	1,125,852	999,975	23,486
_201_compress	18,020,984,917	2,270,140,202	4,274,143,487	16,006
_202_jess	—	—	—	—
_205_raytrace	5,088,752,065	988,420,315	1,231,078,184	109607800
_209_db	9,459,910,842	2,694,222,455	1,664,175,559	443,753
_213_javac	—	—	—	—
_228_jack	5,420,904,450	634,669,636	1,528,148,937	59,361,201

The — signifies that the register allocator failed to compile the program

Figure 6-2: Dynamic Execution Counts, Non-generic Register Allocation

times, and taking the average of the remaining three times.

The instrumentation process breaks the assembly for _227_mtrt, (a multi-threaded raytracing program), so the instrumentation results for the single threaded version, _205_raytrace, have been included as well.

As the numbers show, use of a generic register allocation interface did not result in any inherent deficiencies in the output, or at least no more deficiencies than what any reasonable register allocator might exhibit. As for the compilation times, I contend that the timing differences are due to differences in the algorithms chosen by the authors, not a result of using the new allocation interface. However, justifying this

Spec Benchmark	Generic Register Allocation	Non-generic Register Allocation
_200_check	0.19	0.20
_201_compress	202.23	239.19
_202_jess	374.67	—
_209_db	595.91	601.00
_213_javac	327.10	—
_227_mtrt	282.63	302.68
_228_jack	288.99	314.79

The — signifies that the register allocator failed to compile the program

Figure 6-3: Raw Timing Results (seconds)

Spec Benchmark	Generic Register Allocation	Non-generic Register Allocation
_200_check	5 min 41 sec	4 min 3 sec
_201_compress	5 min 36 sec	3 min 51 sec
_202_jess	11 min 25 sec	—
_205_raytrace	7 min 10 sec	5 min 1 sec
_209_db	6 min 11 sec	4 min 6 sec
_213_javac	26 min 54 sec	—
_227_mtrt	7 min 0 sec	5 min 2 sec
_228_jack	9 min 20 sec	22 min 13 sec

The — signifies that the register allocator failed to compile the program

Figure 6-4: Compilation Times

statement would require sampling a larger space of implementations.

It must be stressed that there are no inherent advantages in terms of execution efficiency provided by a generic register allocation system itself. The difference in performance between the two allocators results from different implementation choices in the allocators themselves, not from the interface design. For example, the generic allocator is more intelligent about evaluating spill costs, by using the best-of-three spill heuristic technique[4].

However, these numbers do demonstrate an extrinsic advantage of using a generic register allocation system: improvements in such a system are shared by all of the compiler backends, not just one. Thus, it is a better use of resources to work on improving the algorithms within a generic register allocator.

In the case of the FLEX compiler, it made more sense for the developer of the StrongARM specific register allocator to work on other projects, because it was more cost-effective to only support and maintain a single allocator.

Chapter 7

Future Work and Conclusions

7.1 Opportunities for Future Research

The abstractions presented here do make many optimizations more difficult (or impossible), since information about the memory model, execution pipeline, and instruction encoding are not expressed in this infrastructure. Even the semantics of the instructions are abstracted away, making it effectively impossible to perform optimizations such as value realization, where instead of spilling a value to memory, the allocator inserts code to recompute the expression generating it on-the-fly. However, there is no inherent reason that the interface could not be extended to communicate such information to the register allocator, as long as the appropriate abstractions were developed. At the very least, it would be nice to extend the interface to support *instruction scheduling*, since, like move coalescing, it is an optimization that can often conflict with register allocation.

Also, it would be worthwhile to explore applying this infrastructure to other architectures. Notably, it would be interesting to see if clever definition of the *pressure* function can allow the allocator to make intelligent use of larger register clusters than the simple pairs presented in this work. The discovery that it was useful for the *pressure* function to be asymmetric came as a shock; perhaps there is potential there for further surprises.

7.2 Possible Drawbacks of Genericity

It is worth noting that the `CodeGen.spec` files are **not** trivial to develop. There are many subtle issues; its not enough to just write a legal assembly string down. The developer often needs to understand variable interference, use/def views, and instr-group semantics. Some difficulty is inherent in maintaining parallel codebases in the `CodeGen.spec` and register allocation classes. One can't put architecture specific debugging code into the register allocator code, so statements like "assert that the two Temps in this Instr don't get mapped to `r0`" can not be placed in the allocator. But perhaps such difficulties are inevitable; compilers are, unarguably, complex beasts.

In any case, verification of a `CodeGen.spec` file might be possible with some sort of exhaustive allocation over all tree patterns. Instead of constructing the interference graph for a program and choosing some allocation that satisfies it, a verifier could construct the interference graph for some minimal program that exercises some small number of patterns, and then try every possible assignment on it. The computational complexity of such a verification seems excessive, since the set of all possible assignments is an exponential function of how many temporaries are involved in the set of patterns. Still, if the number of temporaries used in the patterns is small, this may not be an unreasonable approach.

Another difficulty is based on the fact that not all architectures are created equal, and it is easy to make a register allocator that will produce assembly for one architecture but not another. Therefore, verification of the allocator is difficult, without a least-common-denominator architecture that carries the constraints and intricacies of ALL supported architectures, so that one could test an allocator by targetting that least common denominator architecture. Such an architecture seems unrealizable though.

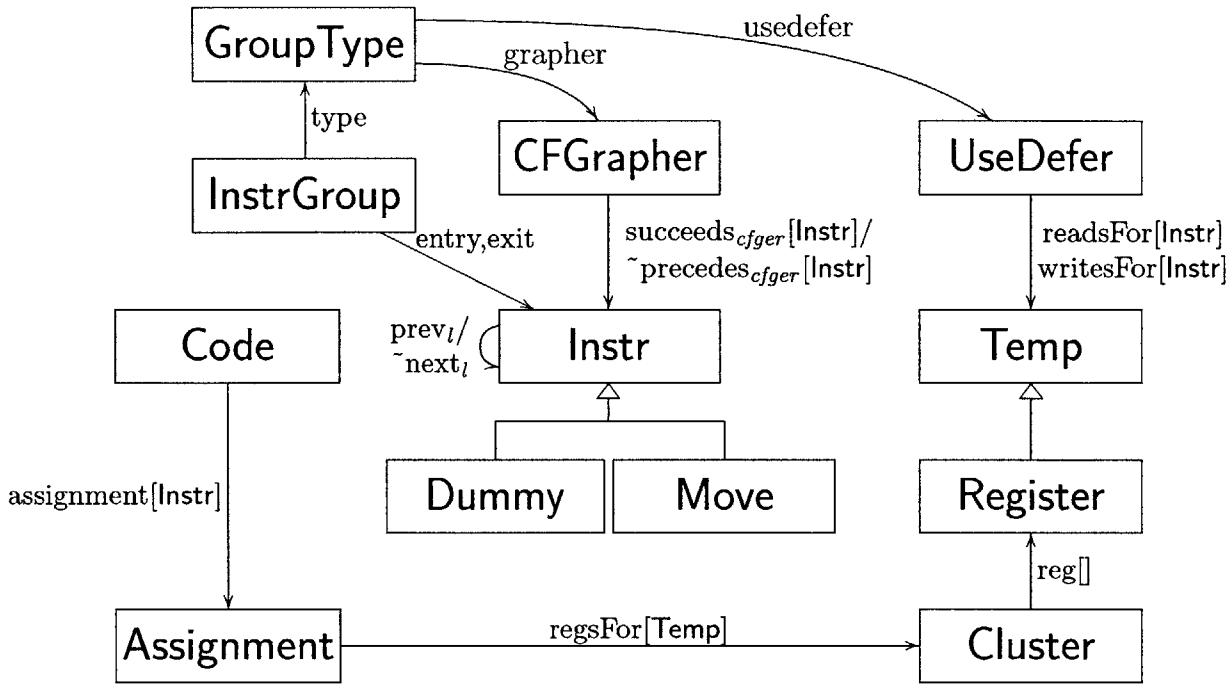

Figure 7-1: Final Object Model for Infrastructure

7.3 The Coalesced Object Model

Over the course of this work, many different extensions have been made to the original simple model for register allocation in order to account for the abstractions introduced in the name of genericity. For completeness, all of the extensions are collected into a single diagram in Figure 7.3.

7.4 Conclusions

A framework can provide an interface for generic register allocation without sacrificing excessive efficiency at execution-time. Of course, the best possible generic register allocator cannot compete with an allocator specialized around a particular architecture; by its very nature, any abstracted allocator will ignore details about the code that a hand-crafted allocator could take advantage of. Similar arguments have been posed in the past against all sorts of generalizations that developers take for granted

today, such as procedural abstraction and high-level languages.

In the long run, abstraction and genericity is a win for compiler developers because it allows for increased reuse of a module and a reduction in programmer effort to support new extensions. Even if a generic allocator is only used as a way to get a series of compiler backends off of the ground quickly, and is replaced by a backend specific register allocator later, the generic allocator will have served a useful purpose.

Appendix A

Register Allocation Interface

Collected here are the signatures of the functions defined earlier in chapter 3. A backend needs to provide a module exporting these functions in order to be integrated with the architecture independent register allocator.

`allRegisters()` :: () → set of Register

All registers available in processor.

`illegal(`t`)` :: Temp → set of Register

All registers which are not part of any legal assignment for t.

`occupancy(`t`)` :: Temp → *Natural*

The number of registers that t can occupy.

`pressure(`t_a`, `t_b`)` :: Temp × Temp → *Natural*

The weight that an interference with t_b should place upon t_a.

`assignment(`t`, `*occupied*`)` :: Temp × set of Register → (Cluster | null)

An assignment for t that uses none of the elements of occupied, or null *if none exists.*

`makeLoad(`*regs*`, `*offset*`)` :: Cluster × *Natural* → list of Instr

Instruction list that loads the value at stack offset offset into regs

`makeStore(`*regs*`, `*offset*`)` :: Cluster × *Natural* → list of Instr

Instruction list that stores the value in regs into stack offset offset

Bibliography

[1] Advanced RISC Machines Limited. *ARM Architecture Reference Manual*, Jul 1995.

[2] C. Scott Ananian. The static single information form. Master's thesis, Massachusetts Institute of Technology, September 1999.

[3] Andrew W. Appel. *Modern compiler implementation in Java.* Press Syndicate of the University of Cambridge, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, UK, 1998.

[4] D. Bernstein, M. C. Golumbic, Y. Mansour, R. Y. Pinter, D. Q. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compilers. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24(7), pages 258–263, Portland, OR, June 1989.

[5] Preston Briggs, Keith D. Cooper, and Linda Torczon. Coloring register pairs. *ACM Letters on Programming Languages and Systems*, 1(1):3–13, March 1992.

[6] Kelvin S. Bryant and Jon Mauney. A Parametric View of Retargetable Register Allocation. Technical Report CS-TR-3403, January 1995.

[7] Charles B. Schoening Carl Mcconnell, J. David Roberts. The rtl system.

[8] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[9] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[10] Intel Corporation. *i960 CA/CF Microprocessor User's Manual*, 1994.

[11] Daniel Jackson. Alloy: A lightweight object modelling notation.

[12] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[13] Brian R. Nickerson. Graph coloring register allocation for processors with multi-register operands. *ACM SIGPLAN Notices*, 25(6):40–52, June 1990.

[14] K. D. Cooper P. Briggs, K. Kennedy, and L. Torczon. Coloring Heuristics for Register Allocation. *SIGPLAN Notices*, 24(7):275–284, July 1989.

[15] R. L. Sites. Machine-independent register allocation. *ACM SIGPLAN Notices*, 14(8):221–225, August 1979.