Register Allocation and Data Conversion in Machine Independent Code Generators

By

Marshall Kirk McKusick

B.S. (Cornell University) 1976
M.S. (University of California) 1979
M.S. (University of California) 1980

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved: Susan L. Graham .... Nov. 12, 1984
Chairman Date

Lawrence A. Rowe .... 14 Nov. 1984

Louis P. Bucklin .... 15 Nov. 1984

....

Register Allocation and Data Conversion
in Machine Independent Code Generators

Copyright © 1984
by
Marshall Kirk McKusick

Register Allocation and Data Conversion
in Machine Independent Code Generators
by
Marshall Kirk McKusick

Abstract

This dissertation investigates the problems of register allocation and intermediate language design in retargetable code generators. The goal is to develop tools that allow high quality code generators to be constructed rapidly.

Formal methods are used to attack the problem of register allocation in a Graham-Glanville table driven code generator. Previous work in register allocation has used a single register allocator following the code generator of the compiler. In this research, the register allocator is split into two phases, a global register allocator for variables and compiler temporaries within a procedure and a local register allocator for temporaries within an expression. The global register allocator is included in an optimizer that runs before code generation. The local register allocator is part of the code generator. Data flow information computed by the optimizer is passed to the local register allocator to suggest which temporaries are good candidates to be assigned to registers. The allocators are table driven so that they can be easily specified and changed.

Experiments were conducted on several multiple pass register allocator algorithms that use graph coloring. Metrics for evaluating the effectiveness of code generators are given and are used to evaluate the register allocation algorithms. Regardless of whether a global optimizer pass is made, the most effective local allocation algorithm improves the running time of a low level language, such as C, by only 1–2%. Higher level languages, such as Pascal, may be improved by as much as 8–10% because the language semantics reduce potential interference from aliasing.

The intermediate language that is passed from a language specific front end to a target machine dependent code generator must be both flexible and compact. General hints are given for the design of intermediate languages. Specific improvements are suggested for the intermediate language used by the UNIX[1] compilers.

---

[1]UNIX is a trademark of AT&T Bell Laboratories.

Table of Contents

Table of Contents		i
Acknowledgments		iii
1	Previous Work	1
1.1	Graham-Glanville Code Generation	2
1.2	Techniques for Register Allocation	4
1.3	The PQCC Code Generation Project	6
1.3.1	Operand Identification and Evaluation Order Selection	7
1.3.2	Instruction Template Matching	9
1.3.3	Register Allocation	10
1.3.4	Code Generation	12
1.4	Evaluation of the PQCC Approach	12
1.5	Overview of the Dissertation	14
2	General Issues in Register Allocation	15
2.1	Partitioning the Registers	15
2.2	Issues in Partitioning Registers Across Subroutine Calls	18
2.3	Caller Save Versus Callee Save	19
2.4	Our Register Model	21
2.5	Some Comments on Benchmark Programs	22
3	Register Management in One Pass Graham-Glanville Code Generators	25
3.1	The Graham-Glanville Code Generator	25
3.2	Register Model in the One Pass Environment	26
3.3	Data Structures Used to Support the Register Model	27
3.3.1	Virtual Registers	27
3.3.2	Register Classes	27
3.3.3	Physical Registers	29
3.3.4	Assignable Registers	29
3.3.5	Spilled Registers	30
3.4	Register Allocation and Assignment	31
3.4.1	Allocation of Temporary Registers	31
3.4.2	Techniques to Avoid Explicit Unspills	33
3.5	Spilling and Unspilling Registers	34
3.5.1	Register Lifetimes	34
3.5.2	Spilling and Unspilling of Code Generator Temporaries	36
3.5.3	Spilling and Unspilling of Tree Transformer Temporaries	37
3.6	Optimizations Within the Current Scheme	37
3.6.1	Using Dedicated Registers as Temporaries	38
3.6.2	Using Other Registers as Backup	38

3.6.3	Detecting Common Subexpressions	39
3.7	Conclusions on the One Pass Model	40
4	Multipass Local Register Allocation Strategies	41
4.1	Structure of Multi-pass Local Register Allocation	42
4.2	Types of Uses for the Registers	43
4.3	Two Pass Register Allocation Strategies	44
4.3.1	Revised Sethi-Ullman Numbering	44
4.3.2	Register Coloring	45
4.3.3	Constraints on Temporary Nodes	50
4.4	Comparison with Other VAX Code Generators	51
4.4.1	Benchmarks of the C Typesetting Program **troff**	52
4.4.2	Benchmarks of the Pascal Typesetting Program **TeX**	53
4.5	Comparison with Other MC68000 Code Generators	55
4.5.1	Benchmarks of the C Typesetting Program **troff**	55
4.5.2	Benchmarks of the Pascal Typesetting Program **TeX**	56
4.6	Improvements Using Dynamic Programming	56
5	Issues in the Design of an Intermediate Representation	59
5.1	Requirements of an IR Design	59
5.2	Problems Encountered with a Production IR	62
5.2.1	Machine Dependencies	63
5.2.2	Machine Versus Language Issues of Addressing	65
5.2.3	General Poor Design Choices	67
6	Data Conversion in Machine Independent Code Generators	69
6.1	Data Conversion Issues	69
6.2	Conversion Handling in Graham-Glanville Code Generators	71
6.3	Optimizing Resource Utilization	72
6.4	Using Semantics to Handle Type Conversion	73
7	Summary and Conclusions	75
References		77

Acknowledgments

I want to thank my parents, Blaine and Marjorie, for all the time and encouragement that they have provided me throughout my lifetime. I also thank them for carrying the heavy financial burden of putting me through Wilmington Friends School and Cornell University. I am grateful to the staff and board of directors of Wilmington Friends School for providing an environment that supported diversity, personal growth, and learning.

Bill Joy was instrumental in convincing me to come to Berkeley, and has been an enormous source of ideas and insights in a wide range of areas during my graduate years. Over the last five years, my companion Eric Allman has provided unwavering emotional support for my efforts. He has also provided amenities normally beyond the reach of a graduate student salary, and the typesetting macro package used herein. I have enjoyed working with Peter Kessler on numerous projects during our graduate student years. He helped make Berkeley conducive to learning as well as pacing me towards finishing my Ph.D.

Susan Graham has provided advising and direction for both my Masters and Ph.D. degrees. Her insights, suggestions, and gentle prodding have been immeasurably helpful in teaching me the art of computer science research. Louis Bucklin and Lawrence Rowe read my dissertation as well as serving with John Ousterhout and Richard Fateman on my research committee. Robert Henry spent countless hours building the code generation system upon which the work in this dissertation is built. He also provided much helpful feedback on drafts of this tome. Edwardo Pelegri-Llopart provided detailed comments on an early draft of this dissertation.

I thank all the people that provided me with financial support while I attended Berkeley. My first year at the University of California was supported by a Berkeley Fellowship funded by the Eugene C. Gee and Mona Fay Gee Scholarship and the Arthur Gould Tasheira Scholarship. During the next three years I received a Howard Hughes Graduate Fellowship funded by the Radar Systems Group under the direction of Charles Runyan. My final years were supported by the National Science Foundation under grants MCS74-07644-A04, MCS78-07291, and MCS-8005144, and the Defense Advance Research Projects Agency (DoD) under ARPA Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

CHAPTER 1

Previous Work

Compilers have been a subject of computer science research since the development of a Fortran compiler by Backus and his group at IBM in the late 50s. Some early work in compilers tried to separate out the issues of language dependencies and machine dependencies [Ershov58], [Strong58], [Steel61]. Pragmatically these ideas were not considered in the design of compilers until recently. Compilers were built monolithically to accept one input language and output code for a single target machine. These compilers had intermediate forms; however the intermediate forms were maintained primarily to help optimization, and were not sufficiently generalized to support multiple languages and target machines.

The portable C compiler developed by Steve Johnson at Bell Laboratories uses a well defined intermediate representation with an external interface. The compiler was designed to promote retargetability to different target machines and was modified subsequently to support multiple language front ends [Johnson77], [Johnson78], [Kessler83]. The flexibility is achieved by separating the compiler into two pieces. The first component is language specific; it is responsible for breaking the source input into tokens, parsing these tokens, and performing semantics checks. The output of the first pass is a language independent intermediate representation. This intermediate representation, or *IR* for short, is composed of expression trees that can be translated to the machine instructions of any of the large variety of available computing hardware.

The second pass is concerned with code generation. It is responsible for reading the IR expressions and finding instructions on the target machine to evaluate them correctly. The second pass should be independent of the language being compiled, and should concern itself strictly with the target machine.

Most Von Neumann architecture machines contain a large main memory with relatively slow access, and a small fast access register file. The registers can either hold data values so that they need not be fetched from the main memory or hold pointers to main memory to reduce the number of bits used in the instruction stream to describe a memory location. An important component of code generators for these machines is a register allocator. The job of the register allocator is to keep the most active data values

in registers to minimize the number of fetches from memory. To produce the most compact possible instruction sequence, the register allocator should use registers as pointers to those data structures such as records and arrays that are too large to keep in registers.

At Berkeley, several researchers with interests in languages and compilers have been studying techniques for machine independent code generators [Aigrain84], [Graham84], [Henry84]. The goal is to ease the task of retargeting compilers by generating high quality code generators from small non-procedural machine descriptions. This dissertation deals with the problem of doing register allocation within the framework of these table driven code generators.

The input to the code generator is the IR produced by the portable C compiler front end. The code generator uses a table driven SLR parser to pattern match the IR to template instructions on the target machine. A register manager allocates the necessary registers for the template instructions selected by the parser and outputs complete target machine instructions. The parser table and the register manager are automatically built from a non-procedural machine description.

The IR design is heavily influenced by its initial use for the C language. It contains dependencies on both the C language and the original target machine, a DEC PDP-11. These dependencies introduce unnecessary complications to the code generator that a more well designed IR would avoid. However, the IR is readily available from front ends for C [Kernighan78], [Johnson77], Pascal [Wirth71], [Joy79], and Fortran 77 [ANSI77], [Feldman79]. Additionally, production code generators are available to translate Johnson's IR to numerous target machines, thus providing a baseline against which to do comparisons with our code generator.

In place of a complete survey of code generation techniques, only the background necessary to discuss the research described in this dissertation is given. The interested reader is directed to the comprehensive book on code generation strategies by Lunell [Lunell83]. A comprehensive summary of the research done in Graham-Glanville and other code generation techniques is provided in Henry's dissertation [Henry84]. The remainder of this chapter gives the background on the Graham-Glanville code generation scheme and on the PQCC work in register allocation. Most of the code generation examples are based on either the MC68000 or VAX architectures [Motorola79], [Digital77].

1.1. Graham-Glanville Code Generation

The main idea in the Graham-Glanville work is the use of augmented parsing techniques to do code generation [Glanville77], [Glanville78], [Graham78], [Graham80]. The target machine is described by productions in a grammar with each production

corresponding to a machine instruction. A standard SLR parser constructor is extended to work with these highly ambiguous machine description grammars. The input to the code generator parser is a preorder traversal of the IR. As the parser does each reduction it emits the instruction associated with the matched production in the grammar. The benefit of using a parser is that it generates provably correct code in linear time using a single deterministic pass.

Glanville's implementation showed the viability of this approach, but could not generate code for the full instruction set of any real machine. Henry improved the Glanville implementation by generalizing the algorithms, speeding up the parser generator, and providing the necessary tools to make it possible to describe real target machines in a practical way [Henry81b].

Schulman produced a production code generator for the VAX-11 using Henry's code generator tools [Schulman81], [Graham82]. Schulman's code generator used hand coded routines for those parts of the code generator that the Graham-Glanville technique did not handle. Some of these routines were executed before the code generator to rewrite IR nodes that the code generator could not handle. For example, on machines without floating point hardware, floating point operations must be replaced with subroutine calls to a floating point library. Other routines were invoked after the code generator to allocate registers to hold the results of temporaries introduced by the IR rewrites and the partially evaluated expressions introduced by the parser.

Henry's dissertation [Henry84] develops more formal machine independent techniques to handle those parts of the code generator that Schulman had to hand code. These formalizations include the initial design for a retargetable one pass register manager described in Chapter 3, and the initial design for a generalized IR rewriting system.

The IR rewrites are done using a top-down left to right table driven pattern matcher. The table is hand crafted; each table entry has three parts. The first part specifies an IR node to be rewritten, the second part specifies a conditional function to be called when a node is matched to determine whether the node should be rewritten, and the third part specifies a routine to rewrite the matched node. The routines to rewrite the matched node were generalized from the primitives used in Schulman's IR rewrites. A more comprehensive study of IR rewrites will be the subject of another dissertation [Pelegri-Llopart84].

A problem inherent in one pass code generation is that it often does not have enough information to do optimal register allocation. In an effort to increase the amount of information available for making decisions about code generation, Ganapathi investigated extending the power of the Graham-Glanville parser by using attribute grammar techniques to add semantics to the basic machine description grammar [Ganapathi80],

[Ganapathi81], [Ganapathi82a], [Ganapathi82b]. These semantic attributes are used to resolve ambiguities in the code selection phase and to propagate register use information within an expression so that the register manager has more complete information on which to make allocation decisions. To retain the speed of the one pass code generation, Ganapathi allows only synthesized attributes to be used. This restriction precludes gathering any information about register uses in future expressions or even unevaluated parts of the current expression. Future use information is important to doing good selection of spill candidates. Admitting the full power of attribute grammars would certainly allow improved register allocation algorithms to be used, though it is probably not possible to do as well as the multipass register coloring algorithms described in Chapter 4.

1.2. Techniques for Register Allocation

Two simple and widely used techniques for improving register allocation are usage counts and Sethi-Ullman numbering [Sethi70]. Even in more complex register allocation schemes such as the one described in this dissertation, variants of both of these techniques are used to assist in doing register allocations. Thus, these techniques are briefly described here, see [Aho77] pages 533-548 for a more complete discussion of the subject.

Register usage counts are predicated on the idea that values that occur most frequently in the body of a procedure can benefit most from being allocated in a register. The static usage counts are usually scaled up when the variable or value is used in a loop in a effort to bias the allocation towards variables that are likely to be frequently accessed. When more complex allocation strategies are used, the register allocator may select several variables as being equally deserving to be placed in the last available register. Usage count strategies can be used to select from among the eligible candidates.

Sethi-Ullman numbering balances an expression tree to reduce the number of registers required to evaluate it. The basic intuition is to evaluate the subexpressions in order of decreasing register usage, so that computed values can be held in registers while other values are computed. In the absence of common subexpressions and hardware that requires the use of register pairs, Sethi-Ullman numbering can determine the evaluation order requiring the fewest number of locations to hold intermediate results. Even if common subexpressions or register pairs are present, Sethi-Ullman numbering can produce a nearly optimal evaluation order. When more complex allocation strategies are used, Sethi-Ullman numbering is frequently applied to produce a first approximation of the evaluation order for each expression.

There are two schools of thought on when register allocation should be done. One approach is to have a single register allocator that runs after code generation. This single

allocation approach is discussed in the next section on the PQCC project. The other approach is to split the allocation between a global optimizer and the code generator. This dissertation uses the split allocation approach.

The advantage of split allocation is separation of tasks. The first pass has the entire text of the program to deal with and is knowledgeable about the semantics of the language. Hence, the first pass has the information needed to make accurate and safe allocations of frequently used variables to the fast hardware registers available on the target machine. The code generator has the knowledge of the target machine needed to assign temporary values to registers.

Unfortunately, split allocation has several drawbacks. First, the front end of a retargetable compiler should be independent of the eventual target machine. If the front end does register allocation, then it must have some knowledge of the eventual target. Secondly, many of the optimizations of register usage are truly independent of the source language being compiled. In developing a retargetable environment, one goal is to reduce the amount of work necessary to write front ends for new languages. If the front end is responsible for emitting optimized IR trees, then the IR optimizations must be done for each new front end that is written. If the optimizations of the IR are done as a later pass, then new front ends are simpler to write since they do not need to optimize the IR trees before emitting them.

To circumvent these problems, current researchers have written high level optimizers that sit between the language specific front ends and the code generating back ends [Chaitin82], [Chow83b]. The front ends are changed to augment the expression trees with the additional semantic information that is relevant to the optimizer. The most important information is notification of potential aliases. Whenever the compiler generates a store through a pointer, it must specify all variables that may be modified. The first pass assumes the worst case, so for a programmer defined unbound pointer (as exists in the C language) it is assumed that all variables in global memory and enclosing stack frames must be changed. Other information includes flagging constructs such as loops (that appear just like any other conditional construct in the intermediate expression trees) so that the optimizer has an idea of where to expect areas of intensified execution.

The global optimizers are responsible for doing flow analysis on the expression trees and for determining the life times of the variables and temporaries generated by the front end. Once this information has been gathered, the optimizer can use it to do transformations on the expression trees to move invariant code out of loops, do strength reduction, etc. [Aho77]. One chore of the optimizer is to find common subexpressions within the expression trees and to pull them out and make them explicit. Finally, the optimizer can assign the most frequently used variables, temporaries, and subexpressions

to registers.

There are several tradeoffs in having the optimizer do these tasks. On the negative side, it must have some knowledge of the target machine, at least to the extent of knowing its register architecture. Having knowledge of the target machine impinges on the philosophy of having a machine independent optimizer. On the positive side, the optimizer need not have much information about the target machine; it really only needs to know how many registers it may use, and the data types that may be placed in them. It already has the other information needed to do the mapping. Specifically, it has the flow information needed to compute lifetimes of the variables that are candidates for being placed in registers and potentially unsafe aliases. Because this information is really independent of the code generation strategies, recoding it for each code generation retarget is unnecessary. Thus, using the same argument that we used for the front end, we push a small amount of information from the code generator back into the optimizer to relieve the burden of having to recalculate (or push forward) all the data flow information.

Because the IR optimizer is target machine independent, it cannot do all the register allocation. Specifically, the optimizer cannot deal with the problems of integrating global register allocation with the local register allocation for the evaluation of expressions because it does not know which temporary values will be computed in registers. Some of the registers must be left for the code generator to allocate. This partitioning leads to the difficult problem of deciding how to divide the registers between the two passes. Despite the difficulty of doing the partitioning, we believe that partitioning is the correct approach as it isolates the language specific, the optimization, and the machine specific code generation problems into well defined interfaces with a minimum of interdependence.

An interesting hardware architecture approach to managing registers is to provide register windows. Examples of this type of hardware are the RISC chips [Patterson81], [Katevenis83], and the Bellmac-32 [Berenbaum82], [Ditzel82]. Register windows allow parameters to be passed in registers, and local variables to be allocated to registers without the cost of register save on routine entry and register restore on routine exit. By eliminating entry and exit costs, the register allocator(s) can more readily use the registers without needing to worry about whether the use will have enough payoff to cover the entry and exit cost. Our register allocation strategies effectively use register window hardware.

1.3. The PQCC Code Generation Project

Concurrently with the development of the Graham-Glanville method, Cattell and others were working on machine independent techniques for generating production quality

compiler constructors, as part of the PQCC project [Barbacci77], [Cattell77], [Cattell78a], [Cattell78b], [Cattell79], [Leverett79], [Cattell80], [Wulf80]. Cattell worked on methods of deriving code sequences based on a sample database of programs. Cattell's idea was to determine code sequences for these programs using axiomatic derivations from a low level machine description. These templates were then used as the basis for code generation. Although Cattell addressed most of the issues of automating code generation, the template construction phase was never built.

In PQCC, register allocation and code generation are done in parallel [Leverett81], [Leverett82]. Leverett uses the template libraries developed by Cattell in a two pass algorithm that does a preliminary pass to estimate register usage, followed by a second pass that generates the code. Leverett believes that all register allocation should be done as part of code generation, since the code generator has complete knowledge of the target machine including the number and types of its registers. It has the necessary information to allocate registers based on the capabilities of the target machine's instruction set. Because instruction selection influences register allocation and vice versa Leverett believes that any register allocation in earlier phases of compilation is going to achieve less than optimal results. If an operator requires one of its operands to be in a register then the code generator can insure that the operand is allocated to a register regardless of whether it is a subexpression, compiler temporary, or user variable.

The goal of Leverett's work is to do register allocation in a retargetable code generator. His dissertation provides a good introduction to the problems involved in building a retargetable register allocator [Leverett81]. Because of the similarity of goals yet divergent solutions between the work done by Leverett and this dissertation, the remainder of this section is devoted to a description of Leverett's register allocation scheme. The next section discusses the shortcomings in Leverett's approach and outlines the solutions developed in this dissertation.

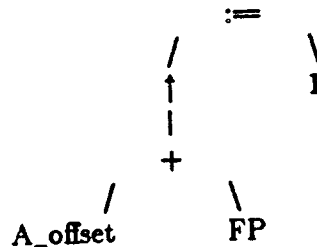
The PQCC code generator is decomposed into four phases. The first phase identifies the addressable operands and chooses the evaluation order. The second phase covers the tree with instruction templates, and identifies needed temporary names. The third phase binds the temporary names to locations. The final phase converts the code templates to real instructions. These phases are discussed in the next four subsections.

1.3.1. Operand Identification and Evaluation Order Selection

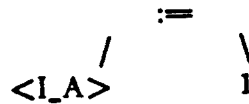
The first phase is divided into three parts, address mode determination, (AMD), combining assignment with arithmetic, (CDEST), and choosing evaluation order, (EVO).

The first part, AMD, finds the computations that can be done by the addressing hardware. The pattern matching is done using a "non-deterministic finite state

automaton" that is never described by either Cattell or Leverett. The branches of the tree matching address nodes are pruned and replaced with leaves showing the set of possible addressing modes. As an example consider the tree for the expression A := 1, where A resides in the local stack frame that is pointed to by the frame pointer register FP.



In this example, the indexed addressing mode would be found for A. The portion of the tree representing the addressing of A would be pruned, resulting in the following tree.



Here <I_A> represents a leaf that describes an indexed addressing mode.

The next part, CDEST, does the ***target path computation***. The goal is to combine arithmetic with assignment by discovering update computations. The target path computation is done by trying to find an instance of each assignment destination as an operand in its source expression. When such a subtree is found, the code generator tries to order the evaluation of the source expression so that the result of the expression can be computed into the destination. At the same time unnecessary pairs of unary operators are removed (UCOMP). For example, "**not not** X" becomes "X".

This computation is best described by two examples. For a VAX, given the statement A := B + C + A, PQCC generates code for A := A + B; A := A + C, using the instructions:

```
addl2  B,A    /* add B to A */
addl2  C,A    /* add C to A */
```

The benefits of doing the target path computation depend on where the target of the computation is located. If "A" is located in memory, then the PQCC code would use 22 bytes of code space and do six memory references. By contrast, the sequence:

```
addl3  B,C,T  /* add B to C, store result in T */
addl2  T,A    /* add T to A */
```

would use only 19 bytes of code space and do four memory references. However, if "A" is

in a register the target path computation avoids the use of a temporary register and saves one code byte.

For an MC68000, given A := B + C (here "B" is in a register since the MC68000 has no memory to memory add), PQCC generates the code:

```
movl  C,A   /* store A in C */
addl  B,A   /* add B to A */
```

An alternative code sequence is:

```
movl  C,T   /* store C in T */
addl  B,T   /* add B to T */
movl  T,A   /* store T in A */
```

Again, the target path computation generates slower code if "A" is located in memory; the PQCC code would use eight words of code space, do four memory references, and use 56 machine cycles, while the alternative code sequence would use only seven words of code space, do two memory references, and use 48 machine cycles. If "A" is in a register, PQCC will save one code word, 4 machine cycles and the use of one temporary register over the other. As these examples show, target path computation is worthwhile only when the destination is in a register.

The final part of the evaluation phase, EVO, chooses the evaluation order. Choosing evaluation order includes tree rewrites (such as adding complements) to insure that the instruction set will cover the tree. Because AMD has already discovered which nodes are covered by hardware addressing modes, a simple Sethi-Ullman numbering algorithm will produce a tree that uses the minimum number of registers to evaluate an expression [Sethi70].

1.3.2. Instruction Template Matching

The second phase consists of instruction selection, the global temporary name allocator, (GTN), and the local temporary name allocator, (LTN). Before beginning code generation, the global register phase, GTN, does procedure-wide flow analysis. The PQCC register allocator considers mapping all user and compiler generated variables into registers. The first step is to identify all programmer defined variables as candidates for placement in registers. Names under consideration include "long-lived" compiler generated variables introduced by the semantics of the language being compiled such as bounds for loop expressions.

The next step, LTN, creates all the "local temporary names". This step involves making a tentative code generation pass to discover all the intermediate temporaries that are needed to evaluate the expression. As intermediate expressions are found they are assigned to temporaries as follows:

1) if matching assignment, use target of assignment.

2) if no restriction on location, use a simple temporary.

3) if restricted (e.g., must be in a register) use two names, T(use) and T(eval). T(eval) has restrictions as needed; T(use) is unrestricted. Specify preference that T(use) be bound to the same location as T(eval).

The primary pattern matcher used for instruction selection is described by four tables:

1) Combined assignment search (equivalent to assigning operators such as += in C)

2) No value search (equivalent to an assignment operator)

3) Flow value search (a boolean that is evaluated only for control flow, not to assign to a programmer variable).

4) Real value search (unary and binary operators such as -, +, etc)

A complex (but machine independent) algorithm selects the appropriate table from which to choose code templates. Table entries with the same operation (e.g., register-register and register-memory versions of an instruction) are merged in a single table entry. Selection of which instruction variant to use is deferred until after register packing is done.

There are several features of this stage of the pattern matching that Leverett notes. By targeting interior nodes of the expression as assignments, the pattern matcher can identify user variables that are best kept in registers. Identifying target variables allows the packing algorithms to combine local and global register allocation. Since addressing modes have been determined and pruned from the tree, the patterns describing the machine operations do not concern themselves with matching addressing operations.

1.3.3. Register Allocation

The four routines in the third phase bind temporary names to storage locations. They are the lifetime analyzer, (LIFE), the register usage estimator, (ESTIMATE), the primary register allocator, (PACK1), and the secondary register allocator, (PACK2). The first routine, LIFE, gathers global (procedure at a time) flow information to determine a conflict graph for temporary names. The next phase, ESTIMATE, determines the minimum number of registers that must be used within the routine.

Once the flow information has been gathered, two packing phases are run. The first, PACK1, assigns registers to temporary names. The basic algorithm is:

```
while temporary_names left {
        rank temporary_names;
        assign top ranked temporary_name to a register
}
```

The ranking criterion is based on two factors:

1) P+ is the savings from having the temporary name in a register versus having it in memory. The savings is computed by multiplying the number of references to the temporary name by the difference in access cost between a register and a memory location. Presumably temporary names in loops are given higher potential savings for being placed in registers, though Leverett never mentions such a weighing criterion.

2) P- is the penalty for having the remaining temporary names in memory rather than in registers. The penalty is computed by taking the average of the P+ figures for the other unassigned temporary names.

The ranking for allocation is calculated based on whether the number of temporary names exceeds the number of free registers. If the number of temporary names is greater than the number of free registers, priority is given to temporary names that must be in a register. If the number of temporary names is less than or equal to the number of registers then the temporary names are ranked as:

P+ – P-

There are two other factors that can modify the rank of a temporary name. If a preference has been indicated to bind two temporary names to the same location through T(use) and T(eval), assignment of one member of the pair affects the ranking of the other. Specifically, the ranking of the other temporary name in the pair immediately rises by the difference of the cost of being allocated to the same location versus any other location. If it cannot bind to the same location its preference reverts back to its old value.

The other factor that can modify the rankings is binding of locations in the instruction templates. Once a location in an instruction is bound, the costs of the other temporary names in the instruction are varied based on the relative costs that are prescribed by the first selection. (e.g., if a register-memory instruction has a temporary name bound to memory then the cost of "load-register" is added to the other operand to preference its being put into a register.)

Once a temporary name has been selected for placing in a register, there are usually several registers in which it could be put. In the BLISS-11 code generator [Wulf75] a back-tracking algorithm selects an appropriate register; Leverett proposes using a non-backtracking voting scheme to select a register. The algorithm traverses each of the

unpacked temporary names that does not conflict with the temporary name being packed; each of these unpacked names gives its P+ vote to those registers to which it could be assigned. Next the preference chain (if any) associated with the temporary name being packed is traversed. One of three states may occur:

1) A temporary name that conflicts with the name being packed is found. The voting stops.

2) A name that has already been packed is found. A P+ vote is given for its location and voting stops.

3) An unpacked name is found. A P- (negative) vote is given to any locations it excludes. Traversal of the preference chain continues.

When the voting is finished, the location with the highest score is selected. PACK1 continues until it has assigned all the registers allocated to it by ESTIMATE.

The assignment of locations for the remaining temporary names that were not handled by the first packing strategy is done by PACK2. The second packing strategy has the option of assigning these locations to either registers or memory, though it is biased against using more registers than suggested by ESTIMATE.

1.3.4. Code Generation

The final phase, (CODE), generates code. All that remains to be done is to generate instructions from the selected code templates and register assignments. CODE moves up the tree selecting the real code to be generated. Despite the efforts of the packing phases, the code generator may still need to allocate "very-temporary" registers. These arise for two reasons:

1) If the instruction requires one register operand and neither operand was packed into a register, a register must be allocated and loaded.

2) A sequence of instructions may require a register in which to do a sub-computation (e.g., the MOD operator on the VAX).

1.4. Evaluation of the PQCC Approach

In his dissertation Leverett claims to have made made four major advances. He believes the most important advance is the combination of local and global register allocation into a single scheme. While having a single register allocator is a desirable goal, he has not fully achieved it as he still needs very-temporary registers. Consequently he still divides up the registers into two groups, though the second group is small. In addition the allocator is complex. The approach of separating the task of global register

allocation and moving it out of the code generator as done by Chow significantly reduces the complexity of the code generator [Chow83b].

Leverett's second claim is that his method improves target path computation since that is not done until addressing modes have been identified. However, the only time that target assignment is profitable is if the target is in a register, a fact that is evident at the time that the decisions is made on which values to place in registers. Hence, the most appropriate time to do transformations for target path computations is much earlier in the compilation process when discovering common subexpressions and doing global register assignment. The code generator can still decide whether update-style instructions are appropriate when the code is generated.

Leverett's third claim is that he balances access costs, preference costs, and template costs to achieve a good heuristic for register allocation. Because he has not implemented his algorithms to date it is difficult to validate or dispute this claim. The apparent shortcoming of his algorithm is that it does not have any metric for assessing the opening cost of a routine. He assumes that opening costs are unimportant. Experience shows that this assumption leads to using too many registers, since in that framework there is no cost for using additional registers [Henry82]. Leverett also does not explain how to deal with register pairs, an unfortunate reality on many machines.

His final claim is that he has devised a voting scheme to predict future needs rather than backtracking when binding registers to temporary names. Again there is no hard data to substantiate this claim. The major shortcoming of his assignment method is that he never considers splitting the lifetime of a variable so that it spends part of its time in a register and part of its time in memory. If a variable is used heavily at the beginning and at the end of a routine, it may well be a candidate for residing in a register. However, it should be possible to spill it to memory during the middle of the routine so that the register can be used for other purposes during that time.

Another problem that arises is that the Compiler Writer's Virtual Machine [Henry84] is not defined. A Compiler Writer's Virtual Machine imposes run time conventions that the compiler must observe, and determines the facilities supplied by the bare machine that the compiler and code generator can use. These conventions include function call linkage, how local and global storage layout is done, and which registers are used as stack and frame pointers. A Compiler Writer's Virtual Machine aids retargetability to the language implementors by hiding the vagaries of the target machines. A common linkage interface provides flexibility to the users of the compilers by allowing them to freely combine programs written in different languages.

There is no provision for rewriting the trees; either the target machine is reflected through to the first pass, or pseudo instructions must be added to cover operations that

the first pass expects that are not on the target machine. Also the pattern matcher is somewhat ad-hoc and is intimately tied to the register allocator. It is spread over four tables (derived from a common source), a finite state machine for address mode computation, and a hierarchical selection among alternate instructions implementing the same operator.

1.5. Overview of the Dissertation

Register allocation is a complex problem even in non-retargetable compilers; generalizing register allocation to encompass multiple target machines and languages adds to these complexities. In this dissertation we modularize the register allocation process to minimize the amount of work required to retarget the code generator to a new machine. We use a strategy of splitting the register allocation task. Global register assignments are handled by a language and target machine independent optimizer and local register assignments are handled by the code generator. The remainder of this dissertation is organized as follows. Chapter 2 outlines the general problems involved in doing register allocation. Chapter 3 describes the framework that we have used to resolve these problems in our table driven code generator. Chapter 4 shows how traditional register optimization techniques can be used in our retargetable code generation technology. Chapter 5 is devoted to a discussion of the problems that were encountered in dealing with Johnson's IR and suggests improvements to reduce its machine and language dependencies. Chapter 6 discusses the issues of type specification in IR's and their realization in the code generator. Chapter 7 summarizes the results presented in this dissertation.

CHAPTER 2

General Issues in Register Allocation

If we are not going to take the approach of doing all register allocation in the code generator, we are faced with the problem of deciding how to partition the available registers. This chapter discusses the alternatives that are available and motivates the partitioning decisions that we use. The final section of this chapter discusses various methods for measuring the performance of register allocators.

2.1. Partitioning the Registers

Certain of the registers in the Compiler Writer's Virtual Machine are usually dedicated to managing the allocation of local variables and temporaries. These registers may be defined by the hardware as on a VAX, or by compiler convention as on the IBM mainframe computers. These registers usually include a top of stack pointer that marks the end of the current memory usage, and a frame pointer that is used as a base register for the local frame. Another register may be dedicated to pointing to the argument list for the routine; more commonly the frame pointer is used for this task as well. Finally several registers may be used to reference enclosing scopes, or these pointers may simply be allocated from the temporary pool as needed. Throughout the remainder of this discussion, register allocation refers to assignment of the non dedicated registers used to hold variables, base pointers, and subexpressions.

The major design decision that must be made is how to divide the registers between the global optimizer and the code generator. One choice is to have a static partition defined as part of the compiler convention. The other choice is to have a more dynamic partition with registers moving back and forth as needed.

The optimal choice is to use a dynamic partition in which the code generator is given just as many registers as it needs and the remainder are given to the optimizer. The reason that the optimal choice is to give the registers to the code generator can be seen by looking at an incremental example. Suppose that we know the number of registers required to evaluate each IR expression. If an expression requires one more register than the optimizer and the code generator have available, then a register must be spilled. If the code generator does the spill, it will be one of the values used in the expression. That

value will then have to be fetched from memory to evaluate the expression; the value will be fetched more than once if it is a common subexpression. However, if the optimizer frees a register, it may be able to spill a value that is not used in the expression. That value may not be used until well after the current expression is evaluated. Meanwhile the freed register can be used to evaluate other expressions. The extra cost to the current expression will be only storing the spilled value. The extra cost to at most one expression (possibly this one) will be the memory reference either to reload the spilled value or to use it from memory.

Unfortunately, the needs of the code generator vary based on the type and complexity of the expressions being computed. These needs are affected by the decisions of the optimizer, since its decision to place an operand in a register may reduce the register needs of the code generator. At a minimum using a dynamic partition would require the code generator to be run twice, once to find out its expected register needs for each expression before the optimizer is run, then again after the optimizer has done its work [Karr84]. The number of iterations in the worst case is equal to the number of registers to be allocated. Because of the high cost of these two phases running them iteratively is unacceptable.

An alternative to dynamic partitioning is to use a fixed partition. A fixed partition allocates a fixed number of registers to each pass. Using a fixed partition lessens the dependence of the two passes on each other, but can lead to less efficient usage of the scarce register resources. If the code generator has too few registers it will be forced to generate spill code. Often the marginal improvement in the last register allocated by the optimizer is less than the cost of the generated spill. Conversely, if the code generator has extra registers that it does not need, they remain idle while operands that could profitably have been put into registers by the optimizer remain in memory.

One solution to effectively use the registers is to only provide the code generator with a few registers. The optimizer does a Sethi-Ullman numbering on the expressions, and reorders or breaks up the expressions that are too complex to be evaluated without spills by the code generator. The code generator would be able to evaluate the simplified expression without needing to generate any spills. The optimizer could then assign some of its extra registers to hold the results of the intermediate temporaries that it creates, thus reducing the number of implicit intermediate results kept in memory.

The problem with this technique is that the optimizer must now have more knowledge about the target machine. For example, it must know how to find addressing modes at the leaves of the expression trees. In addition, for machines that require certain operands to be in registers, it must consider that even though a subtree represents an addressable operand it will still need to be loaded into a temporary register. While

certain machine specific information such as how to do constant arithmetic on the target machine is necessary, having a complete machine specific pattern matcher to probe for address mode determination and instruction register needs is excessive. An alternative is to use a conservative approach and assume that all operands will have to be evaluated by machine instructions, but this assumption will usually result in unused registers in the code generator.

The second possibility is that the optimizer can leave register allocation of common subexpressions that are wholly contained within a single *code unit* to the code generator. Here a code unit is a computation that is expressed by the language specific first pass as the larger of a basic block or a single expression tree. Such a tree may contain implied control flow such as short circuit evaluation or an implied if-then-else evaluation for value. These trees typically require several temporaries to track the result of the computation, yet have well characterized control flow enabling decisions to be made without requiring complete flow analysis.

The basic idea is that these common subexpressions will be assigned to temporary nodes in the expression trees passed to the code generator. If the code generator has registers remaining unused after generating code for the basic block, these extra registers can be used to hold the value computed by the temporary node. If there are insufficient registers to hold all the temporary nodes, the register allocator decides whether it is better to compute the common subexpression once and save it in a memory location, or if it is better to recompute the common subexpression each time that it is used. Of course, placing a value in a register rather than in memory or recomputing it can result in another register being freed. So the process has to iterate until all the registers are used, or until no further subexpressions are available.

Another alternative to effectively use the registers is to try to put the values of all common subexpressions into registers and only choose to put some in memory if the register allocator runs out. This approach still requires multiple passes since the register manager must select a common subexpression value to put in memory when it runs out of registers. Putting a common subexpression value in memory may in turn require all code following the first use of the common subexpression to be regenerated because it depended on having the common subexpression value in a register. The coloring algorithms discussed in Chapter 4 attempt both approaches.

This solution does give the ability to provide uses for the unused registers assigned to the code generator. The problem is that basic blocks are usually short and typically contain few if any common subexpressions. Thus, there are no candidates for using up the code generator temporaries. It would be possible to expand the scope of consideration of subexpressions to span code units, since the optimizer has already found them while

computing the flow information. However, expanding the scope of consideration can result in significant backtracking as the code generator must first generate code for all the blocks to discover whether it will have at least one free register throughout all the blocks that can be devoted to the subexpression. Once having done this, it will need to iterate over all the basic blocks again to regenerate code that may result in another free register, etc. Taken to an extreme, this approach is little better than having the movable register partition.

In our experience we have found that single code units are the most reasonable size to consider. A single code unit is the largest unit that the code generator local register allocator can allocate without global flow information. Common subexpressions that span single code units tend to encompass many code units and hence the code generator must greatly increase the number of expressions considered before many new possibilities for register allocation become available.

2.2. Issues in Partitioning Registers Across Subroutine Calls

Another major decision that must be made is how to handle the lifetimes of the allocatable registers in the context of subroutine calls. There are two major alternatives. The first alternative is to save all registers across subroutine calls. The registers may be saved at either the point of the call, or at the entry to the call; these alternatives are discussed in the next section. Saving all registers provides the greatest flexibility in allocating the registers since they are all available for allocation without regard to subroutine calls. However, this strategy also incurs the highest cost since every register that may be used within a subroutine must be saved at entry and restored at exit.

A second alternative is to assume that all registers are destroyed across subroutine calls. Here the entry cost of routines is low since they need not save or restore registers. However, the usefulness of registers is greatly reduced since their lifetimes cannot span subroutine calls.

Although the first alternative has traditionally been the protocol of choice, an approach using a hybrid of the two alternatives has been used to good effect in recent compilers [Johnson78a]. In the hybrid scheme the registers are divided into two roughly equal sized groups. One group is saved across routine calls, and the other group is not saved. The registers that are saved are used to hold values that are typically live through a large part of the routine such as loop indices, programmer variables, etc. The registers that are not saved are used to hold common subexpressions and intermediate values. These temporary registers typically have short lifetimes. This scheme allows routines to have the benefits of long lived registers yet still have registers for scratch computations without the entry and exit costs.

2.3. Caller Save Versus Callee Save

Another related decision that must be made is what protocol to use for saving and restoring registers across subroutine calls. There are at least three options. One alternative is to save registers at the point of call. Here the register save code must be generated to save all active registers before the routine call, and restore the registers after its return. A second option is to save registers at the entry point of the routine. The called routine need only save the registers that it can potentially modify. Saving registers at routine entry has traditionally been the protocol of choice since the register save code appears only at the single entry point rather than being duplicated at the possibly multiple call sites. The final option is to compute dynamically which registers to save [Steele80]. As part of the runtime subroutine linkage, the set of registers to save is computed by taking the intersection of the set of live registers in the calling routine and the set of registers used in the called routine.

None of these protocols is optimal in all cases. Saving registers at the entry point is optimal if the calling routine uses more registers than the called routine, since fewer registers need to be saved at the entry point than would need to be saved at the call site. Conversely, saving registers at the call site is optimal if the called routine uses more registers than the calling routine, since fewer registers need to be saved at the call site than would need to be saved at the entry point.

While the dynamic linkage appears to save the minimum number of registers in both of these cases, in reality it quickly degenerates to an expensive form of callee save. The hidden cost lies in the set of registers that the caller must specify as live to the called routine. This set includes the union of any registers that it is using plus any registers that were specified by its caller that it did not save. If any routine in the dynamic call chain uses all the registers, it will pass forward a set that requests that all registers be saved. Thus, even routines that use few registers themselves will still present a save set that specifies nearly all the registers. The intersection of a set specifying all the registers, with any set specified at the callee, is just the callee's set. Hence, this scheme degenerates to callee save. The higher expense arises from the computation of the set of registers to be saved that must be done at runtime for each call. Up to two additional memory stores must be done to record the set of registers really saved, and the set of registers not saved. These extra memory stores further erode any potential gain.

Minimization of register save and restore costs requires the use of interprocedural analysis to select the optimal calling convention on a call by call basis. Some interprocedural analyzers now run in linear time based on the size if the program, and can update their information incrementally as the program is changed [Cooper84]. However, there are still instances in which separate compilation occurs, for example, when the

program must link to external libraries that are available only in binary form. Use of parametric routines also requires a standard interface, since interprocedural information cannot always determine which routine(s) will really be called. Consequently, the issue of how to allocate registers across separately compiled routines will never become a moot point.

Our production compilers currently save registers at procedure entry. In an effort to evaluate the cost tradeoffs between the two calling conventions, we modified our compiler to save registers at the calling site. No other modifications were made; the production register allocation strategy was used. Because the production compiler does not do any dead variable analysis, all registers used in a procedure were saved before each subroutine call. We recompiled all the UNIX[1] utilities using the modified compiler. Statically we found that the text of the program grew by about 8%. To get a metric on the dynamic cost, we installed the recompiled utilities on one of our research machines and collected accounting data over a two week period. We compared the average running time of those utilities that accumulated at least ten seconds of running time and were run at least ten times. Of the 22 utilities that fell into this group, four ran measurably slower, three ran measurably faster, and the rest showed no change. The biggest running time improvement was 5%, the biggest running time loss was 4%. The differences were small and primarily a function of the style in which the programs were written.

On the assumption that it will not be possible to choose dynamically which register save convention to use on a call by call basis using a global analyzer, a fixed policy must be selected. Because the dynamic costs between the two strategies are similar, and the static costs are lower for saving registers at routine entry, the saving at entry strategy appears to be better. However, when registers are saved at routine entry there is no simple way to measure the incremental value of using one more register within the routine being compiled. The major benefit of using the save at point of call strategy is that it provides a much better metric for measuring the cost of procedure entry and exit. When considering whether to put a value in a register, the cost of all the register saves and restores can be calculated by looking at the call sites contained within the routine for which code is being generated. This information gives the optimizer a handle on the cost/benefit tradeoff in assigning an expression or variable to a register since it will know exactly when and where the registers must be saved and restored. Since the saves and restores are embedded at the call site for which code is being generated, the code generator has the opportunity to avoid explicit restores by using values directly from the save locations. Finally it allows the distinction between saved and temporary registers to

[1]UNIX is a trademark of AT&T Bell Laboratories.

be blurred. The routine can simply arrange to save any registers that are live at the point of call.

As an example consider a complex routine whose body requires the use of all the registers. If it is compiled using the save at entry point strategy and this routine has a conditional as the first statement that returns from the routine on 90% of the calls, it will do many unnecessary register saves and restores. If flow information is available it may be possible to avoid the restores. However, most machines have hardware support for subroutine linkage and the runtime cost of code to cleanup the stack frame without restoring the registers may be higher than using the hardware return instruction(s) that restore the registers.

If on the other hand the routine uses the save at point of call strategy, the only time that this routine will ever save and restore registers is in the 10% of the cases where the body really executes and does other subroutine calls. The strategy is not without its degenerate case in which a routine with many active registers calls a simple routine that immediately returns. Even though the save and restore were unnecessary for the subroutine call, the save and restore cost is still cheaper than not having the values in registers or presumably the optimizer would not have done the assignment. In either case, the routine would be more efficient if the complex conditional body were a nested routine.

A drawback of using the save at call site strategy is the extra code space it requires. In a general timesharing environment, memory size is increasing more quickly than processor speed. Thus, given a time/space tradeoff, we nearly always optimize to minimize the time. In a general timesharing environment, the increase in speed when using a register allocation strategy that can capitalize on the added information that saving at call site provides is worth the added memory cost.

Unfortunately, general time sharing is not a universal model. With the advent of personal work stations, processor cycles become much more readily available though memory may need to be paged over a network. Thus, many of the time versus space decisions may need to be reevaluated. For example, the decision to use a calling convention that increases the text size may prove to slow down the actual execution because of the extra paging. The methodologies outlined in this dissertation should provide a useful starting point, even if the actual decisions are different.

2.4. Our Register Model

As in the UNIX compiler convention, the registers in our model are divided into two fixed partitions; one partition is called *dedicated* and the other *temporary*. The dedicated registers are allocated by either the language specific front end or the language and machine independent optimizer that is run before the code generator. All uses of the

dedicated registers are managed by the passes preceding code generation, hence all references to these registers appear explicitly in the IR trees input to the code generator. The front end is responsible for generating all spills and loads of the dedicated registers including those necessary across subroutine calls. Either register save convention can be used; we use the save at point of call convention. Thus, the code generator need not concern itself with reference counting these registers.

The temporary registers are not saved across subroutine calls and are allocated by the code generator. These allocations fall into two broad categories. The main use of temporary registers is to hold partial results as they are computed during the process of evaluating an expression. The other category of temporaries are derived from the generation of common subexpressions and intermediate results because of low level rewrites of the IR trees. The major low level IR rewrite is to pull function calls out of expressions. All instances of function calls embedded in an expression are evaluated first and stored into temporaries. These temporaries are then put into the expression at the points from which the function calls were removed and the expression is then evaluated. These temporaries appear in the IR as temporary nodes. The register allocator decides whether each temporary node is allocated into a register or a memory location.

2.5. Some Comments on Benchmark Programs

In discussing different register allocation schemes one must come up with some metric for comparing their effectiveness. These comparisons are all too often composed of a few perverse expressions that push on limits in the code generation system. The implication is that if the code generator can handle the perverse cases well, it will have no trouble dealing with the simpler expressions that are normally encountered. Unfortunately, generating locally optimal code is often done at global expense. A register allocator that manages to get everything into registers may really cause a program to run more slowly because it fails to account for procedure opening costs and overuses the registers.

To alleviate this problem, benchmarks are often used to measure the global effectiveness of a register allocator. Benchmarks are at best an approximation to reality. While it would be desirable to have a small set of programs that characterize all the programs that are typically run on any given system, finding such a set of programs is an impossible goal [Weicker84]. Rather, than attempt to find a small set of programs contrived to illustrate a point, the entire set of utilities provided with 4.2BSD is used. Most of these programs are written in C, and hence suffer the biases of C programmers. Undoubtedly if a large body of Fortran programs were used, the results would be significantly different [Knuth71]. However, improvements to this set of programs is

certainly useful to a large user population.

To provide a set of "complex expressions" to use as test data, all the utilities were compiled, and all the expression trees that did not cause our simple one pass register manager to use more than two register temporaries were discarded. Discarding the simple expression trees eliminated 97% of the expression trees and 93% of the expression nodes (the remaining expression trees were typically more complex than those that were discarded). When running various register allocation schemes, statistics were gathered by running all these complex expressions through the code generator and then calculating the average effectiveness of the result. Many of these expressions could have been rewritten at the source code level to be functionally equivalent but simpler and faster. The rewritting option was not pursued.

In evaluating the overall effectiveness of different register allocation policies, it is not enough to compare the results of register allocation on random bits and pieces of expression trees. Ideally we would like to gather dynamic statistics by recompiling all the utilities and running them for two weeks in our user environment. Several factors prevented us from doing so. The biggest problem was getting the necessary resources. Compiling all the utilities using our fastest one pass compiler requires eighteen hours of VAX 11/750 time. Some of the register allocation schemes increase the compilation time by a factor of ten. There is the problem of finding enough two week slots in which to run the experiments, and of finding a user population willing to tolerate having usually stable utilities crash after two hours of use because of a compilation bug. Finally the early version of the global optimizer provided to us by Hennessy and Chow did not have a front end for C. Thus, to run C programs through it requires augmenting the output of our production C front end by hand to add the necessary information.

Given that it was impractical to use this method to test all the register allocation ideas that we have had, we used a faster and simpler approach to screen out all but the most promising strategies. As a vehicle for experimentation we selected the utilities that represented the biggest use of CPU time on our system, the programs that did typesetting. These programs were compiled with different register allocation strategies and their running time was measured on a sample document. The register allocation strategies that appeared most promising were used to recompile the entire set of utilities which were then measured in a two week usage experiment.

The standard typesetting utility, **troff**, was written in C. It was run through the C front end and the output hand crafted so that it could be run through the Chow IR optimizer. The optimized and unoptimized IRs were then used as input to a code generator configured to use the register allocation strategy under test. The size of the executable binary output by the code generator measured the static effectiveness of the

register allocation strategy. The time taken to typeset a 19 page entry from the user's manual measured the dynamic effectiveness of the register allocation strategy.

The other typesetting utility, **TeX**, was written in Pascal. It was run through Chow's Pascal front end, that provided all the necessary information needed by the optimizer. Both optimized and unoptimized IR's were then input to the various code generators configured to use the register allocation strategy under test. The effectiveness of the executable binary output by the code generator was determined by measuring how long it took to typeset a thirty-two page manuscript that contained many mathematical equations and special font requests.

The next two chapters discuss the various register allocation strategies that were tried. Because many of the strategies were never tested beyond the simple test described above, all the tabulated results are for the simple test so that they may be readily compared. The most promising allocation policies require the use of the global optimizer. Unfortunately, the global optimizer with a C language front end has not been finished at the time that this dissertation is being written, hence the results of the full tests are not included.

CHAPTER 3

Register Management in One Pass Graham-Glanville Code Generators

This chapter begins with a description of the Graham-Glanville code generator that we use. Next we discuss the model that we use to describe the characteristics of the registers available on a target machine. The basic register management algorithms are based on well known technology. Next it describes more sophisticated algorithms used to manage the register resources. Finally it concludes with a discussion of the problems in doing one pass register allocation and motivates the multipass algorithms described in Chapter 4.

3.1. The Graham-Glanville Code Generator

Before starting code generation, the IR trees are rewritten to eliminate operators that cannot be implemented by a single instruction on the target machine. One rewrite that is done in an attempt to reduce register requirements is to reorder commutative operators to place the child with the larger number of nodes as their left operand. This heuristic, a replacement for Sethi-Ullman numbering, assumes that the subtree with more nodes will require more registers for its evaluation. By placing the larger tree on the left, it will be evaluated first when there are more registers available.

After all the IR rewrites have been done, the lexical analyzer for the IR does a left-to-right, top down traversal passing each node to the code generator as it is first encountered. The code generator runs as a coroutine, consuming tokens as they are produced. As the code generator matches productions in the machine grammar it calls semantic routines that either build an address, or emit an instruction.

The temporary registers allocated by the code generator fall into two broad categories. The main use of temporary registers is to hold partial results as they are computed. These results occur as the interior nodes of an expression are evaluated by the emission of an instruction by the code generator. The other category of temporaries, derived from the low level IR rewrites, appear in the IR as temporary nodes.

The initial implementation of the code generator uses a strictly one pass code generation algorithm with no backup. Thus, all register allocation is done synchronously with the emission of instructions. Temporaries are not assigned to physical registers until

they are first encountered by the code generator in the emission of an instruction that uses them. The purpose of the delay is to put off making an allocation decision for as long as possible. This technique prevents a register from appearing busy before it is really in use, thus allowing the most up to date possible information to be used in making an assignment decision. If a source operand is in a register that is being used for the last time, the register manager chooses to use that register as the destination register for the instruction. The reuse of a source register for the result often eliminates the need for a temporary register load on a two address machine, or allows the peephole optimizer to change a three address instruction into a two address instruction on a machine with both two and three address instructions.

3.2. Register Model in the One Pass Environment

The register manager exports a model of sets of typed *virtual registers*. Each set of virtual registers models a datatype on the underlying machine. Thus, if a machine has data types of word, long, and float, the register manager provides three sets of virtual registers that can contain values of each of the three types. The set of virtual registers for each data type is further broken down into two subsets corresponding to the dedicated and temporary registers available for the type.

These two subsets are managed differently. The virtual registers in the dedicated register set are always mapped directly to their corresponding physical registers. There are exactly as many virtual registers as there are physical registers. Because the dedicated registers appear explicitly in the IR, the IR nodes can be translated directly to their corresponding virtual registers. Because these registers are managed entirely by the earlier phases, the register manager does not maintain reference counts, or generate any loads or spills. If two different register types share the same set of physical registers, the front end is responsible for insuring that live variables do not overlap the same physical register.

The temporary registers are handled much differently. The virtual registers are mapped to physical registers or appropriately sized memory locations depending on the availability of resources at the time that the virtual registers are used. There is no limit to the number of virtual registers that can be allocated to any particular type, if the number of live virtual registers exceeds the number of available physical registers on the machine, some of the virtual registers are spilled. Because physical register management is handled internally by the register manager, any explicit use of a temporary physical register in the IR is treated as an error.

When a temporary virtual register is allocated it must specify the number of future uses. Each time that the virtual register is subsequently used, its use count is

decremented. When the count reaches zero, the virtual register is freed. The set of live virtual registers is determined by scanning the number of virtual registers with a non-zero future use count. The register manager allows virtual registers of different types to reside in the same set of physical registers. Thus, registers of type word and long may use the same set of physical registers on the machine. The register manager insures that assignment of physical registers is unique across all types of temporary virtual registers. Doing a unique assignment can be complicated by a virtual register that is bound to a pair of physical registers, since it may have to spill two virtual registers of a type that uses only a single physical register before it can begin using the register pair.

3.3. Data Structures Used to Support the Register Model

There are five basic data structures used to support the register model. The relationship of these data structures is shown Figure 3.1. Each of the data structures is described in a following subsection.

3.3.1. Virtual Registers

Virtual registers are the interface between the code generator and the register manager. They are obtained by the code generator in one of two ways. When a register node corresponding to a dedicated register is encountered in the IR, the register manager is requested to provide a virtual register bound to the specified dedicated register. When a temporary register is needed, the code generator obtains a virtual register from the register manager specifying only the data type to be placed in it, and the number of uses to which it will be put.

The register manager always returns a virtual register pointer. This pointer is carried around as the semantic information associated with the register. All the information about the allocation state, contents, location, and use count is maintained in the virtual register structure. Because the semantic routines may copy the virtual register pointer arbitrarily, the register manager cannot count the number of references to the virtual register. It simply assumes that it can deallocate the structure when its use count drops to zero.

3.3.2. Register Classes

The register classes identify the machine types available in the target machine. There is one class for each machine type. The set of types available on the target machine defines the data types available in the Compiler Writers Virtual Machine; the set of classes is specified for each retarget to a new architecture. Associated with the class are the size and alignment requirements for memory locations to be used if a physical register

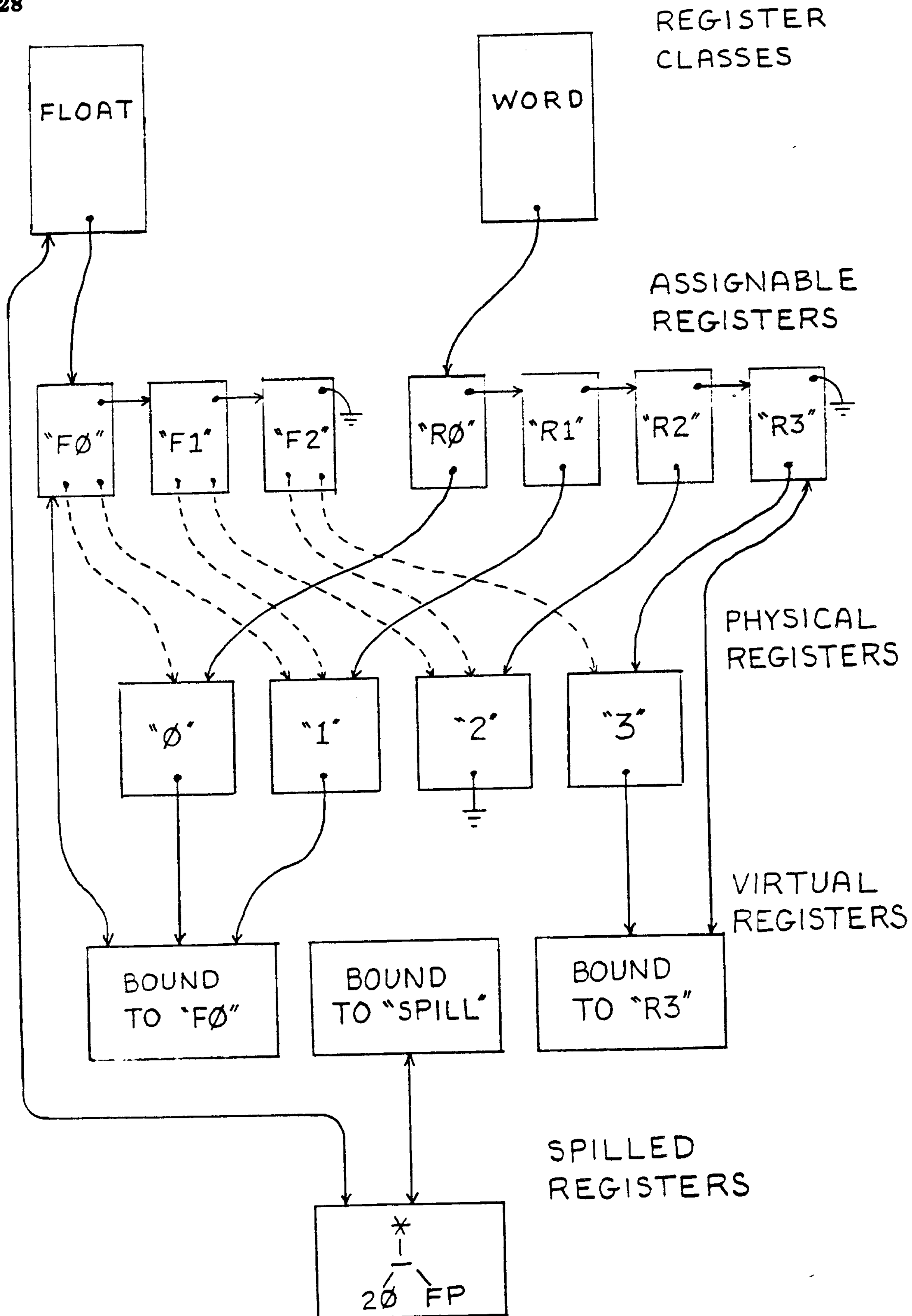


Figure 3.1 – Data structures used to support the register model.

associated with the class must be spilled.

The class structure contains a pointer to a list of all the possible physical registers that may be used to hold values of its type. This set of registers includes both dedicated and temporary registers. The list must contain all the dedicated registers that can appear in the IR of the type specified by the class. If the IR never specifies registers of the type associated with the class, then the possibility of dedicated registers need not be specified.

Similarly the class must contain at least one temporary register if the code generator will ever need to allocate temporaries in that type class. The exception to this rule occurs if the architecture supports *storage inspecific* operands; these operands are usable for data types whose instructions can always use a memory operand interchangeably with register operands; floating point on the VAX is an example. For data types with storage inspecific operands, the register allocator will allocate virtual registers in memory if there are no temporary physical registers allocated to the class, or if none of the physical registers allocated to the class are available without spilling.

3.3.3. Physical Registers

In addition to describing the data classes on an architecture, a retarget to a new machine must describe each of the available hardware registers. Each register on the machine is described exactly once, even if it can be used to hold more than one data class. Thus, a general purpose register on the VAX may be used to hold the value of a scalar or a floating point value. Note that all classes usually will not map to all physical registers. For example, the PDP-11 has both general purpose registers and floating point registers. While all these registers require a unique structure, only the general purpose registers are listed in the scalar classes, and only floating point registers are listed in the floating point classes.

The purpose of having a single structure per physical register is to arbitrate its use if the register can be claimed by several different classes. When a physical register is assigned to a virtual register, the physical register structure is removed from the free list and made to point to its virtual register owner. When the register manager is scanning the list of possible registers for a particular class, it can determine if a physical register is available even if it is being used as part of a different class.

3.3.4. Assignable Registers

The final structure that must be described for each retarget, classifies the physical registers into their appropriate classes. Each class has a list of all the possible physical registers that can be used to hold values with the data type associated with the class. The list consists of a chain of assignable register structures; there is an assignable register for

each physical register that is suitable for use by the class. A physical register may be claimed by assignable registers from several different classes. For example, if the general purpose registers can contain both word and long data types these two classes will have assignable registers that cover exactly the same set of physical registers.

Some data classes such as double precision floating point on the VAX may require two or more registers to hold a value. Thus, an assignable register may point to an array of physical registers. These arrays of registers need not be disjoint. Thus, if the first assignable register points to register pair r0 and r1, the second assignable register can point to register pair r1 and r2, the third assignable register can point to register pair r3 and r4, etc. When the first assignable register is bound to a virtual register, the physical register structures for both r0 and r1 are removed from the free list and set to point to the virtual register that owns them. Thus, if the register manager is called on to allocate another register pair of the same type, it will skip over the second assignable register as it will find that one of its associated physical registers is in use.

Associated with each assignable register is a set of attributes that describe the purpose for which the register can be used. The most important attribute is whether the register is dedicated or temporary. This attribute is dynamic since the register manager may choose to move registers between the two partitions during code generation. This attribute is checked whenever a register node is encountered in the IR to insure that the requested register is marked as dedicated. Similarly only registers marked as temporary may ever be used by the register manager. Other attributes show the assignable register that holds the return value from a function call. The register manager exports this knowledge by providing an interface that allows the code generator to request the function return register by class without knowing the physical register used for that purpose. The final attribute that is maintained for each assignable register is a string that describes how to print out the register in assembly language. This attribute is maintained at the assignable register level rather than at the physical register level since some machines use different names for the same physical register depending on how the register is being used.

3.3.5. Spilled Registers

The final data structure used by the register manager describes the location of the spill area. When a virtual register is spilled to free its associated physical register for other uses the physical register is unbound from the virtual register and is replaced by a structure describing the location of the spill area. Usually the spill areas are allocated in the runtime stack, however, on non-stack architectures the register manager may be provided with a global temporary area. When a spilled virtual register is accessed during

an instruction paste-up, the register manager uses the spill structure to locate the value associated with the spilled virtual register.

The register manager currently maintains the spill areas by class type. Classifying spill areas by type may result in two different spill locations being allocated for two different types of values, even though they have disjoint lifetimes and identical sized memory requirements. A more economical use of spill locations could be obtained if they were organized by size of the spill areas instead of by types.

3.4. Register Allocation and Assignment

The register manager is responsible for assigning the available physical registers to the virtual registers used by the code generator. The assignment of dedicated registers is simple, since the front ends are assumed to do all the lifetime analysis to insure that the dedicated registers never contain values with overlapping lifetimes. Hence, the register manager assumes that the virtual register is always bound to its associated physical register. The only exception to this rule occurs when a dedicated register has been loaned to the pool of temporary registers (see section 3.5.1). Borrowing entails spilling the old contents; when the register is next encountered as a dedicated register from the front end it must be either reloaded from the spill area or replaced with a reference to the spill area. Tracking the lifetime of temporary registers is the responsibility of the register manager.

3.4.1. Allocation of Temporary Registers

The allocation of registers is exported from the register manager through the virtual register interface. The first step is to allocate a virtual register descriptor. The virtual register descriptor is obtained by calling an allocation routine specifying the type, use count, and special needs that are required. The type must be a type specified when the classes were defined. The special needs can be used to request a specific register such as "function return register for this class", or may even specify a particular register by number. The register manager does not allocate a particular register at the time that the virtual register is allocated, but simply notes all the requirements. All further requests to the register manager use the virtual register descriptor that the allocation routine returns.

The uses of the register are driven by the instruction paste-up routines. When the code generator wants to emit an instruction, it calls a paste-up routine with the operation and a list of the operands to use. The operands are either virtual registers, or combinations of virtual registers and offsets packaged up into more complex addressing modes. The paste-up routine passes each of the operands to the register manager in turn, showing that they are about to be used. The register manager is responsible for insuring that the virtual register is bound to a physical register and for returning an assembly

language string that describes the physical register.

The action taken by the register manager depends on the state of the virtual register. On the first use of the virtual register, it has never had any physical register assigned to it. If the virtual register requires a particular register, the required register is found and freed by spilling its contents if they are live. If a spill is generated, the virtual register whose contents were spilled is unbound from the physical register and bound to the spill area. The physical register is then bound to the virtual register being allocated and its assembly language print name is returned. If any register in the class can be used then the least cost register available is located, spilled if necessary, and allocated to the virtual register. A register is selected by looking at all the assignable registers associated with the class and selecting the one with the least cost. The selection criteria are described in the next section.

When the register manager is called with a virtual register that has been previously used, it checks to see if the virtual register is still bound to a physical register. If it is still bound to a physical register, the register manager simply decrements the use count for the register and returns its assembly language string. If the virtual register is bound to a spill area the register manager takes one of two actions. If the virtual register is in a class that allows storage inspecific operands, the register manager returns an assembly language string that references the spill area.

If the register is in a context that may require the contents to reside in a physical register, the only way to insure that an instruction can be emitted is to always reload the value into a physical register. The reloading can be done in the same way as the initial allocation policy. If a specific register is required then it is allocated, otherwise a least cost analysis is done on all the registers in the class. The selected physical register is freed by spilling if necessary, and is then reloaded from the spill area associated with the virtual register.

The register manager handles the last use of a virtual register, shown by a use count of one, slightly differently. If the virtual register is resident in a physical register, the virtual register is marked as being *in transit*. If the destination operand of the instruction is a register being allocated for the first time, the register allocator prefers to allocate an in transit source over any other register. This allocation strategy reduces the number of operands on three address machines, or register copies on two address machines by making a source be the same as the destination. Finally the register manager deallocates the resources associated with the virtual register and puts its associated physical register back on the free list.

3.4.2. Techniques to Avoid Explicit Unspills

Often spilled values can be used directly from memory without being unspilled explicitly into a register. One way to get that information is to modify the parser constructor that builds the code generator to augment each state that reduces a register addressing mode to show whether a memory based addressing mode can be substituted. When doing instruction emission, this memory usage information is passed to the register manager with each virtual register that is contained in the instruction. If the register manager determines that the contents of the virtual register resides in memory, it can avoid generating an unspill when the memory usage information shows that the virtual register is used in a context that allows a memory operand.

One problem with this optimization arises on non-orthogonal machines such as the MC68000 in which the instruction set allows either one but not both of the operands to be in memory. For non-orthogonal machines, the state information is dependent on other operands; it is not possible to determine whether the first operand can be replaced with a memory based operand without knowing how the second operand is used. This problem is mitigated on the MC68000 because it is a two address instruction machine, thus the second source is also the destination. Because of the high cost of having both the source and destination in memory the parser generator can statically decide that the second operand should always be unspilled into a register, and thus that the first operand can always be accessed from memory.

Another problem with the augmenting method is that it must know the addressing modes that are needed to access temporary locations. The amount of state information is dependent on where temporaries may be allocated for the target machine. The table generator must produce a machine dependent sized array of state showing the suitability of each possible type of temporary. Semantics must be written to interpret the state by determining the type of the temporary and then checking for the appropriate information. Consider the MC68000 with two types of register temporaries, and both word and long displacement memory locations. The register temporaries would always be addressable, the word displacement sometimes addressable, and the long displacement never addressable.

To avoid these problems we have chosen to introduce a one instruction backup. The pasteup routine that is responsible for emitting the instruction is informed by the register manager that one or more of its virtual registers has been spilled. If a spill has occurred, the pasteup routine constructs an IR tree corresponding to the instruction that it was about to emit. The nodes corresponding to the spilled registers are replaced with the trees representing the spilled memory locations. This tree is run recursively through the code generator. If the target machine has provision for using the operand directly from

memory it generates the single appropriate instruction saving an explicit load instruction and the need for an extra temporary register. If the operand really needs to be in a register, the code generator allocates and loads a temporary register and then uses it in the register form of the instruction.

The benefit of using this scheme over having the parser augment the states of the parse table with additional information is that it uses the information that is already available in the parse tables. The drawback is that building a new IR tree and reentering the parser is slower than having all the needed information precomputed. We have chosen to use the limited backup because it works using an existing mechanism, and because the semantic routine that unbundles an instruction to its IR is easily derived from the address mode description part of the machine grammar.

3.5. Spilling and Unspilling Registers

The crux of the register allocation problem is contained in the register allocation routine that must decide which registers to spill. A register allocator that can determine a near optimal solution cannot be implemented in a one pass code generation scheme because of lack of future use information. Our initial work focused on doing the best that we could in a one pass environment, since for development environments the speed of the code generator is more important than producing optimal code. The rest of this chapter discusses the one pass policies that we tried. The next chapter discusses the more sophisticated algorithms that we use when the code generator is not constrained to run in a single pass.

3.5.1. Register Lifetimes

The first step is to define a policy for choosing the registers to spill. The policy that we use is to spill registers that have the furthest future use. The furthest future use policy minimizes the number of loads and stores if all computations must be carried out in a single type of register and there are no common subexpressions. However, this policy can be suboptimal if certain operations can be computed directly into memory, or draw their operands from memory without an intervening load. In these cases it may be faster to avoid spilling an operand that may have the furthest future use, and cannot be reused from memory in favor of one that will be used sooner, but can be used from memory. Similarly it may be more advantageous to compute an intermediate result directly into a spill location rather than into a register that must be spilled. These calculations require multiple passes, hence for the one pass code generator we chose to use the furthest future use policy.

Given this policy one must come up with some implementation of it. To motivate the heuristic that we used to approximate the furthest future use policy, we need to consider the three types of registers that are handled by the register manager. The first type are the dedicated registers; since the register manager has no control over the dedicated registers it can ignore them. The second type are the temporaries produced as the result of partial evaluation by the code generator. Because the code generator is driven by a parser, and the parser pushes its partial results onto its parse stack, the order of usage of the intermediate results is strictly "last in, first out". Under such a usage pattern, spilling the "least recently used" register insures that the register with the furthest future use will be spilled.

The third type of register usage is from the common subexpressions generated by the rewrites of the IR preceding the parser. To accurately determine the temporary that has the furthest future use would require one of two alternatives. The register manager could collect future use information. Alternatively it could guess and then regenerate code by rerunning the affected IR trees through the parser to generate new code when the guess proved wrong. Both these choices are precluded by the one pass restriction on the code generator. Thus, we must use some predictive scheme. Many of the temporaries produced as a result of rewriting the IR have clustered usage patterns. A "least recently used" strategy does well at differentiating between those that are clustered and those that are not. Hence, we chose to use a "least recently used" strategy for all our temporary allocation. This strategy always yields furthest future use for code generator temporaries; it works acceptably well with temporaries generated from IR rewrites that tend to have clustered usage patterns.

A common instance of common subexpressions arises from evaluating an if-then-else construct for value. These subexpressions are difficult to handle in one pass code generation. Since the code generator does not know if it will have a spare register to hold the common subexpression in every block until it has generated code for every block, it cannot allocate this type of common subexpression to a register. To avoid the difficulty of restoring the state of the registers after each block (by reloading any registers that were spilled during code generation for the block), the register allocator always spills all active registers when entering a multiple definition, single use scope.

The least recently used strategy can be implemented cheaply. At the time that a physical register is bound to a virtual register a global "lifetime generation number" is incremented and assigned to it. Each time that the virtual register is used in the emission of an instruction, the physical register associated with the virtual register is assigned the next lifetime generation number. The allocation of a new register is done by scanning through the list of assignable registers in the appropriate register class. While doing this

scan the allocator keeps a pointer to the oldest register in the list; the oldest register is the one with the smallest lifetime number. If the register class requires the use of register pairs, the lifetime numbers of each of the registers in the pair are summed and the register with the oldest sum is saved. If free registers are set to have a lifetime generation number zero, then a single walk of the list of assignable registers will produce the best register to use. If the lifetime value is zero the register is free. If it is non-zero, then the register is live and must be spilled.

3.5.2. Spilling and Unspilling of Code Generator Temporaries

Once the register manager has selected an assignable register that it wants to allocate to a virtual register, it requests the spill manager to save any live value(s) contained by the physical register(s) associated with the assignable register. The spill manager does this task by walking through the set of physical registers associated with the assignable register and checking to see if they are owned by any virtual register. If they are owned, then the contents are spilled and the spill location is bound to the virtual register.

As an example, consider that the register manager decides that it wishes to assign a double precision register pair associated with physical registers r0 and r1. Suppose also that r0 is free, and r1 is bound to a virtual register associated with a long integer. The register manager would then spill r1 into a long integer in memory, and then return r0 and r1 bound to the virtual register containing the double precision value. If the long integer was next used while the double precision value was still active, the register manager would probably choose to unspill it into some other register such as r2. However, if it chose to unspill the long back into r0 or r1, both registers in the register pair would be spilled as a double rather than spilling only half of the value as is minimally required.

To avoid having machine dependencies in the spill manager it does not generate instructions to spill registers directly. Instead, it constructs an IR tree that assigns the virtual register to the spill location. This IR tree is passed back to the code generator to be processed just like any other IR tree. Thus, the full instruction set repertoire is available for generating the spill.

As described above this strategy can result in spilling both registers in a pair when only one is needed. It can also result in generating two integer spills to free each of the registers of a pair, when a single move of the register pair would have worked as well. In practice we have found that these less than optimal moves occur infrequently, and that the peephole optimizer can usually patch up the two integer move case to use a single instruction.

The unspilling of a value works in much the same way as a spill. The register manager selects a least cost assignable register to unspill the value into and passes it to the spill manager. The spill manager inspects the set of physical registers, spilling any that are active as above. It then binds the physical registers to the virtual register that is being unspilled. Finally it generates an IR tree that assigns the spill location to that virtual register and hands the tree to the code generator to have the code emitted.

3.5.3. Spilling and Unspilling of Tree Transformer Temporaries

The temporaries that are allocated by the IR rewrites are placed in the IR as a special new leaf type, TEMP. The names of all live temporaries are maintained in a temporary table by the register manager. The temporary table contains a pointer to a virtual register associated with the temporary node. When the lexical analyzer encounters a TEMP leaf in the IR it looks up the temporary in the temporary table. The associated virtual register is handed to the register manager for an allocation. Unlike the registers allocated for intermediate results that must always reside in a hardware register, TEMP nodes are treated as storage inspecific. Thus, the register manager may return a memory location for the temporary leaf if all the hardware registers are in use or if the virtual register associated with the temporary has been spilled. If the virtual register corresponding to a TEMP node is in memory then the IR tree corresponding to the spill area is fed to the code generator instead of the virtual register. Handing the IR tree to the code generator allows it to consider emitting an instruction that uses the value directly from memory rather than first doing an explicit unspill. If the temporary is used in a context in which a hardware register is required, an ensuing allocation and load will occur. No backtracking is required since the code generator is encountering the particular use of the temporary for the first time.

The table holding the TEMP node to virtual register mappings are reference counted with the same reference count as the TEMP node's virtual register. When the TEMP node is encountered for the last time, both its virtual register and its table entry are reclaimed.

3.6. Optimizations Within the Current Scheme

There are several optimizations that could be made within the context of the current one pass allocation scheme. They have not been implemented within the one pass context since they require a significant amount of complex code to do in the one pass context. They are all easily handled by the multipass organizations; so long as they are implemented at least once, we have the opportunity to measure their effectiveness at improving the code generator.

3.6.1. Using Dedicated Registers as Temporaries

When faced with a shortage of temporary registers, the register manager could commandeer an unused dedicated register rather than using a spill location. The biggest problem with this strategy is in determining whether the benefits of using a dedicated register outweigh the costs. The costs of using a dedicated register are that it must be saved on routine entry and restored on routine exit. If the expression that requires a temporary is in a piece of the routine that is rarely executed, the entry/exit cost is unlikely to outweigh the infrequent dynamic cost of doing the spill. Except in the rare case in which the expression is not preceded by any possible routine exits and is not contained in any conditional code, the only method of determining the appropriate cost tradeoff is to have run time profiling information, or programmer hints.

Several logistical problems also arise. The code generator must now buffer up an entire routine rather than just an expression at a time to determine whether any of the dedicated registers are really available. If the code generator elects to use one or more of the dedicated registers, it must arrange to save and restore those registers that it uses. Because the register save/restore code is usually generated by the language specific front end, the code generator writer must know how to interpret and modify or augment the entry code.

3.6.2. Using Other Registers as Backup

Many machines have alternate locations that can be used as temporaries that are faster to access than the standard memory in the local stack frame location. These locations include other types of registers or fast scratchpad memories. Examples of other types of registers include unused scratch floating point registers. Typically the architecture cannot support scalar operations in these registers, but they can frequently be loaded and restored from the general purpose registers more quickly than a memory location. Thus, if an operand must be spilled and particularly if it cannot be reused from memory, saving and restoring it from a register is more effective than the normal memory spill.

Another example of register backup comes from the MC68000 data and address register architecture. If a data register is needed when only an address register is available, it is much faster to spill a data register to an address register rather than using a memory location. Even if the operand could be reused from memory but cannot be used from the address register, using a scratch register is an improvement since two register to register copies are faster than even a single memory store.

Less desirable than spare registers, but still somewhat beneficial is the use of memory that can be cheaply accessed using some form of abbreviated (and hence faster) addressing

method. Examples include the global registers on the RISC machine, or the first 65,536 bytes of memory on the MC68000 machine. If these resources are available they are managed in the same way as other scratch registers.

The main requirement to managing scratch locations is to be able to know when they may be used. Since these locations are visible to all routines and are not saved across subroutine calls, the code generator must take care to identify when they can and cannot be used. Thus, if a location is being spilled because a subroutine call is about to be generated, then the spill must go into the local stack frame. In general the temporary manager must check the lifetime of each temporary that it allocates to insure that its lifetime does not span a subroutine call. To avoid calling doing these checks as a second pass, they can be done as the IR is read in and buffered in preparation for code generation. As the IR for the routine is read in, all subroutine calls are discovered. Temporaries with a definition before a subroutine call and a use after a subroutine call are not considered as candidates for being held in registers because they would always have to be spilled. The spill is particularly bad when the value could have been calculated into the local stack frame in the first place.

3.6.3. Detecting Common Subexpressions

The most general case of common subexpressions requires global flow analysis to determine when variables are defined, used, and killed. However, common subexpressions within a basic blocks can be easily found and exploited. Consider for example the C expression that assigns the field of a record:

```
p->f1 = p->f2;
```

that generates the VAX code:

```
movl  p,r0
movl  p,r1
movl  f2(r1),f1(r0)
```

Here the common subexpression corresponding to the pointer value held in "p" could be held in a single register resulting in the simpler code sequence:

```
movl  p,r0
movl  f2(r0),f1(r0)
```

Finding a concise way to track the contents of registers is difficult. Also, maintaining common subexpressions increases the complexity of the spill algorithm. The decision on the optimal register to spill becomes more dependent on future usage patterns that are difficult to characterize.

3.7. Conclusions on the One Pass Model

This register model has been flexible enough to describe several different register based machines. These machines include two reduced instruction set machines that have register windows, the complex but orthogonal VAX instruction set that has a uniform set of general purpose registers, and the non-orthogonal MC68000 that has two functionally overlapping sets of registers.

The register manager does better than one would expect given its naive algorithms. When compiling the entire set of UNIX utilities on the VAX, it only has to generate spill code for ten expressions. The crux of the success is that most expressions are simple enough that the register manager never has to spill any registers. Thus, the only loss is that some of the registers sit idle when they could profitably be put to use. When spilling does occur, the furthest future use algorithm works well. Its most notable failure occurs when temporaries get used in a round robin order; this usage pattern causes the register allocator to constantly spill and unspill values.

The next chapter discusses how improvements to this simple strategy can be made given the opportunity to do multi-pass code generation.

CHAPTER 4

Multipass Local Register Allocation Strategies

This chapter introduces the problems in doing multi-pass register allocation for local temporaries and common subexpressions. It then describes several coloring strategies for solving these problems. It concludes by comparing the effectiveness of these strategies and making suggestions for further optimizations.

The task of register allocation is not handled entirely by the multi-pass code generator. Machine independent optimizations, including assignment of programmer variables to registers, is done by a separate pass that precedes the code generation pass [Chow83b]. The only target machine information available to the optimizer is the number and types of registers available for it to allocate. Among other tasks, the optimizing pass is responsible for allocating and assigning the dedicated registers to programmer variables and common subexpressions. Often the optimizer has more candidates for registers than it has registers in which to place them. These candidates for registers are marked so that the code generator can use any unused registers in its temporary partition to hold them rather than storing them in memory locations.

The task of the register manager in the code generator is to allocate the intermediate results of expression computations into the registers in its temporary partition. If there are temporary registers left unused after allocating registers to hold intermediate results, they are used to hold the common subexpressions identified but not allocated to registers by the optimizer. The lifetime of registers allocated by the code generator is typically constrained to a single code unit. Running the optimizer is optional; however, the code generator will not attempt to take over any of the allocation tasks usually performed by the optimizer such as allocating the dedicated registers to user variables. Furthermore, without the common subexpression information provided by the optimizer, the code generator is not able to use up any left over registers.

One pass strategies are limited by their lack of future resource allocation information. One approach to getting future use information is to have a global flow analyzer calculate the future information that the code generator will need. Then the one pass strategy would have all the information that it needed to make reasonable decisions. In practice, precalculation of flow information does not work effectively since the

information that the code generator requires is not known until the instructions are selected. The reasons that the global flow analyzer cannot supply all the future information are the same as the reasons that it cannot do all the register allocation. It would require knowing too much about the target architecture.

4.1. Structure of Multi-pass Local Register Allocation

To collect future use information, a multi-pass code generation strategy is used. The simplest multi-pass strategy uses two passes. The first pass collects information on register needs and uses and the second pass uses this information to assign registers and allocate the minimum number of spills and reloads. Because the spills and reloads may affect the information collected by the first pass it may be necessary to run additional passes to converge to an optimal solution. In practice we find that the solution converges in two passes in almost all cases. In the cases in which it fails to converge, the code that it does produce is acceptably close to the solution that it finds if it allowed to run to convergence. The bound on the number of passes can be increased for those instances in which the improvement is deemed worth the cost in extra code generation time.

The actual implementation of the multi-pass approach iterates over the code generator. On the first pass the register manager hypothesizes an unlimited number of registers. The code generator then produces code using as many temporary registers as it needs. Rather, than really emitting the code, it simply annotates the IR tree with the instructions that it would generate given an unlimited number of registers. Multiuse temporaries are calculated into registers and then held in those registers until their last uses. Because there are an unlimited number of temporary registers, there are no spill or reload instructions generated. The only memory references are those caused by actual operands.

For most expressions the naively generated code needs fewer registers than really exist. In these cases the code selected during the first pass does not require any registers to be spilled and the second pass can be skipped; the code is emitted by a walk over the expression tree to dump out the generated code sequence.

For expressions that require more temporary registers than are available, some temporaries with non-overlapping lifetimes must use the same physical register. The process of sharing the same physical register is called *merging*. If the merging process cannot reduce the number of temporaries to the number of available registers some of the results have to be spilled. The job of the second pass is to decide which temporaries should be spilled and which temporaries assigned to registers. The techniques for making these choices are discussed in the next section.

Once a set of selections has been made, the expression trees are modified to reflect the choices. If a subexpression is to be spilled, the IR tree defining the subexpression is pruned from the main IR tree and used as the right hand side of an assignment to a spill location. The spill location to which the pruned subexpression is assigned replaces the subexpression in the tree.

The forest of expression trees is then run through the code generator for the second time. If the register strategy is limited to two passes, then the register manager is set to allocate only as many registers as are really available so that it can generate spills and reloads (using its one pass approach). The spills and reloads may occur if the allocation choices affected the code generation in a way that requires it to use more registers. For example, if an operation requires its operands to be in registers and one of its operands has been spilled, the code generator will need an extra register in which to load the spilled operand before it can generate the operation.

If the register strategy is using an unlimited number of passes then the code generator is run as before with an unlimited number of temporary registers. Once again, it annotates the expression tree with the instructions that it would have generated. If the number of temporaries exceeds the number of available registers the same or possibly some new allocation strategy is run to reduce the number of registers required. This cycle is iterated until a code generation pass is run that has register requirements within the number of available temporaries, or until an upper bound on the number of passes is reached. In the first case a tree walk is done to emit the generated code. In the second case the code generator is forced to use its one pass strategy for handling the tree as described above.

4.2. Types of Uses for the Registers

Once a tentative code generation has been done, all the temporaries that will be required by the code have been made explicit. These registers include the intermediate results computed during the evaluation of the expression, as well as registers allocated to common subexpressions by the optimizer or the tree rewriting phases that precede the code generation.

The registers allocated to hold intermediate results are used in a strictly LIFO order. Thus, the lifetimes and overlaps of these registers are known trivially. The common subexpressions do not follow such a simple usage pattern.

The common subexpressions fall into two categories. The first category follows the typical pattern of having a single definition, followed by several uses. These expressions usually arise as part of an address calculation or as part of a complicated expression such as a field extraction or insertion. In attempting to map such expressions into registers

there are two considerations. If a register is free from the definition to the first use, then it may be worth keeping the expression in the register even though it will need to be saved and restored for later uses. The exception occurs when the cost of computing the expression is less than the cost of saving and restoring it. Here it is better to simply recalculate the expression before it will next be needed.

The other category of common subexpression is one that has multiple definitions but only a single use. These occur when doing short circuit evaluation, or evaluating an if-then-else construct for value. In this instance one or more conditionals branch out into a set of basic blocks that must then evaluate some appropriate expression. All the basic blocks compute a value and branch to a common point; at the common point the value is either assigned to a variable or used as the basis for a conditional branch. The tricky step in handling this case is that the state of the registers must be reset before evaluating each basic block, and the state must be restored on exit from the block. Furthermore, if a register is assigned to hold the value of the resulting common subexpression, that register must be used in all the definitions.

4.3. Two Pass Register Allocation Strategies

Once all the temporaries have been identified, some method of assigning the registers to them must be used. Each of the three different types of temporaries requires a different allocation strategy. Rather, than try to develop many special purpose types of allocation strategies, A more generalized allocation technique is used that fulfills the needs of all the different allocation requirements.

4.3.1. Revised Sethi-Ullman Numbering

Once a tentative code generation pass has been done, the operations that can be handled by the addressing modes have been partitioned from the set of operations that must be handled by explicit machine instructions. All the intermediate results are shown explicitly as temporary nodes in the tree. Thus, it becomes possible to run a straightforward Sethi-Ullman numbering to rearrange the tree to reduce the number of temporaries that are needed. Because the addressing modes have been pruned from the tree (based on the target machine), the Sethi-Ullman numbering need not make estimations of the capabilities of the target machine, yet can still make accurate estimations of temporary needs. If the tree is rearranged then it is rerun through the code generator to select new instructions.

Given the multi-pass algorithm, doing Sethi-Ullman numbering during the IR rewrites preceding the first attempt at code generation may not seem worthwhile. However, the approximate Sethi-Ullman numbering does the right set of transformations

in 94% of the expressions in our test suite. Hence, the cost of an extra code generation usually can be avoided. That significantly improves the running time of the code generator. Typically the only time that the second Sethi-Ullman numbering finds any improvements are cases in which a complex addressing mode that is matched in the tree consumes more nodes than were estimated by the tentative Sethi-Ullman numbering. In these few cases the second Sethi-Ullman numbering is able to reduce the number of required temporaries and is worth the extra cost of the code generation. If the tree still required more temporary registers than were available, an algorithm such as the local coloring described in the next section determines loads and spills. If the number of temporaries required is reduced to the number available, then no further work needs to be done other than to walk the tree and emit the code.

4.3.2. Register Coloring

In an effort to find a good register allocation, Chaitin uses an interference graph to represent overlapping lifetimes of variables over an entire routine. The register allocator then tries to color the graph, where the set of chromatics are equal to the number of available registers [Chaitin82]. The first step in using the coloring algorithm is to create a dependency graph among the different variables. Each distinct variable is represented in the dependency graph by a node. The edges in the graph represent overlapping lifetimes or dependencies. Thus, if two variables are live at the same point they are said to be dependent on each other and have an edge between their respective nodes. Register allocation is done by finding a coloring for the graph. A successful coloring is one in which no two nodes with a connecting edge have the same color. If the number of colors required does not exceed the number of hardware registers, then a successful allocation has been achieved.

If the problem can be solved, then no spills need to be generated. If variables must be spilled, the objective is to minimize the number of dynamic memory references. Since we cannot measure the number of dynamic memory references without having run time data, we assume that minimizing the static references will also minimize the dynamic references. In an effort to improve the static information, the use count for variables are multiplied by ten for each level of loop nesting. These assumptions are validated by gathering run time data on the various spill algorithms that we use.

Finding a minimal coloring is an NP complete problem; thus for even moderate sized problems heuristics must be developed to find approximate solutions. Experiments with three different coloring heuristics were conducted. The first heuristic is the one suggested by Chaitin. He assumes that he is restricted to a set of colors equal to the number of available registers. If all nodes in the graph have fewer incident edges than the number of

available colors, then the graph can be trivially colored. If there exist nodes that have greater than the available number of colors as incident edges, then the node with the greatest number of incident edges is assigned to a memory temporary and removed from the graph. Nodes are removed in this way until no node contains incident edges greater than the available number of colors. Coloring can then proceed as described above.

Chaitin's work is intended for the IBM-801, a machine that has numerous registers, so he rarely has to resort to spilling [Radin82]. Unfortunately, his algorithm for choosing nodes to spill, while rapidly removing edges from the graph, also tends to remove variables that are live over large parts of the program. These variables may simply be defined early and used at the end, or they may be used extensively throughout a routine. Chaitin's scheme does not account for this disparity, and can have negative consequences if it is applied frequently. We concur with Chaitin that it works best when there are many registers so that the spilling is seldom needed. Since we are specifically trying to deal with the cases in which some values must be spilled, this is a poor assumption.

We have modified Chaitin's algorithm slightly to try to differentiate between frequently and distantly used variables. Each node in the graph is augmented with a count of the number of uses of the temporary in the code being examined. The selection of a node to be removed from the graph is done by first finding the N nodes with the greatest number of incident edges. The node in this group with the fewest number of uses (independent of the number of edges) is selected to be removed.

To understand the intuition behind this algorithm, consider two variables each with a node in the graph. Suppose that one variable is heavily used in a local area of the routine, while the other is defined at the start of the routine and used once at the end of the routine. Both of these variables will have many incident edges because of interference with other variables; the frequently used one because of its many uses, the distantly used one because of its long lifetime. This algorithm will be more inclined to spill the distantly used variable than the frequently used variable.

To test out the modified Chaitin algorithm, the test set of complex expressions were run through the coloring algorithm giving it three general purpose registers for allocation. The average expression in the test set has 6.3 temporaries, each accessed 2.6 times. The average number of temporaries that were spilled and the average number of static memory accesses per expression that were generated were computed using different values of N. Note that when N is equal to one my algorithm is equivalent to Chaitin's original algorithm. Figure 4.1 shows the results.

N	avg # spilled temps	avg # static mem refs per expr
1	2.2	8.5
2	2.7	7.3
3	3.0	7.8
4	3.7	9.2
5	4.1	9.3

Figure 4.1 – Optimization of memory references in graph coloring.

For $N = 2$, selection by number of uses is beneficial since the nodes have roughly equal numbers of incident edges. Consequently removing either node serves to reduce the complexity of the graph significantly. After spilling an average of only 2.7 nodes, the graph can be colored. As the number of nodes under consideration increases, the number of uses dominates and nodes with few edges are removed. So, although the node removal does not introduce many memory references, it also does not significantly reduce the complexity of the graph. By the time that five nodes are considered, 4.1 temporaries must be placed in memory before the graph can be effectively colored.

The optimal value of N is highly dependent on the number of colors that are available for use and the number of nodes in the graph. Thus, the actual variant of the algorithm that we use tries to dynamically select values of N based on available resources. If the graph cannot be colored, the set of nodes to be considered for spilling is initialized to be the node with the greatest number of incident edges. Each of the remaining nodes in the graph is inspected. If its number of incident edges is at least 80% of the number of incident edges of the node with the greatest number of incident edges, it is added to the set of nodes under consideration for spilling. The value of N is the cardinality of the set of nodes under consideration. The 80% value is derived empirically by trying different values on the set of complex expressions. The algorithm works equally well for values between 65% and 90%, thus 80% was chosen as being in the middle of this range.

Using this algorithm for selecting N, the coloring was rerun on the same set of complex expressions. On average the algorithm had to remove three nodes from the graph before it was able to color it. Figure 4.2 shows the frequency that each pass was run, and the value of N in each pass. Each pass corresponds to selecting a node to be spilled, thus 85% of the expressions required at least two nodes to be spilled. The value of N shows the number of nodes that were in the set to select a spill candidate from. This algorithm allocated an average of 2.8 temporaries using an average of 6.9 static memory references. These figures are better than those for the first method.

Pass #	frequency run	N
1	100%	2.2
2	85%	4.0
3	46%	3.3
4	19%	2.7
5+	3%	2.3

Figure 4.2 – Values used for N for each temporary selection.

This heuristic has the benefit that it runs quickly, and finds a nearly optimal register assignment in most cases. Unfortunately, it does not fail gracefully, for certain examples it generates far more memory references than are necessary.

In an effort to apply Chaitin's work to more conventional architectures, Chow has developed a different heuristic for dealing with the resolution of spills [Chow83b]. The strategy proceeds by first assigning colors to the most promising nodes in the graph based on the number of memory references that they avoid. When the coloring becomes blocked, such that there is no way to color the remaining uncolored nodes, the algorithm resorts to node splitting. Node splitting partitions the lifetime of a temporary into two smaller partitions. The temporary is saved in a memory location during the period between the two partitions. The effect of partitioning the lifetime of a temporary is that it usually reduces the number of dependencies in the graph, reducing the number of incident edges to many of the uncolored nodes. The allocation and splitting proceeds until all the nodes are either colored or not worth coloring (for most machines, if a partition is so small that it only has a single use, the cost of loading a register for the one use is not worthwhile).

The heuristic in this algorithm involves deciding which lifetime to split and where to split it. The primary strategy is to find a cut point in the graph that has a minimum number of active crossings, thus minimizing the number of save and restore operations that are introduced. This algorithm works well in the context of an entire procedure, however, in the smaller context of a single code generation unit the choice of which lifetime to split is less clear.

The failure of the heuristic is caused by the peculiar nature of the temporaries. Except for the one definition-multiple use temporaries, splitting a lifetime always results in relegating it to memory. Because the single definition-multiple use temporaries are typically the most desirable to color, the node splitting reduces to deciding which single use temporaries to place in memory. No heuristics were found to bias the heuristic to work as well as the variant of Chaitin's described above. The problem can be summarized by noting that it is easier to decide which temporaries belong in memory and put the rest into registers than it is to decide which temporaries to put into registers and put the rest

into memory.

To provide a metric to compare the heuristic strategies, an exhaustive enumeration of all possible register assignments was implemented. There are several approaches that can be taken. The first enumeration method assumed that lifetimes could not be split. Each temporary lived either entirely in a register, or always in memory. Thus, this enumeration model produced the optimal graph that the Chaitin coloring heuristic could hope to find. The algorithm starts with a number of colors equal to the maximum number of incident edges to any node plus one and proceeds to try every possible coloring scheme until a valid coloring is found or all the possibilities have been tried. If all possibilities have been tried without success, an additional color is added and the search is run again. This process is repeated until a valid coloring scheme is found.

Once a valid coloring scheme is found, the actual register allocation must be done. If the number of colors does not exceed the number of available registers, an arbitrary selection can be made. In the more usual case in which the number of colors exceeds the number of available registers, the method selects nodes that have the minimum number of uses to be allocated to memory temporaries.

A comparison of the exhaustively enumerated results with those of Chaitin and myself are shown in Figure 4.3. Keep in mind that the complex expressions reported below only represent the 7% of code units that caused spills in the one pass code generation scheme. If all code units are included, all the methods place temporaries in registers in more than 99% of the allocations.

Method	avg # mem ref	% temp access in mem	# mem refs per mem temp	% temps in mem
Pure Chaitin	8.5	52%	3.9	35%
Modified Chaitin	6.9	42%	2.5	44%
Temp Enumeration	6.4	39%	2.1	48%

Figure 4.3 – Utilization of registers for temporaries on the VAX with three registers available for allocation.

The conclusions from these tests are that the modified Chaitin heuristic works well. However, for small code units the running time of the exhaustive enumeration is competitive. It usually runs faster than the Chaitin method, with a few notable exceptions. One alternative would be to use a heuristic used by database systems in which a time limit is set and the exhaustive enumeration started. If the time limit is reached before a solution is found, the code generator would use one of its heuristics. Note that using an exhaustive enumeration works well only because this problem is typically small. This solution would not be appropriate for the larger scope of problem

that Chaitin originally solved.

Chow's method allows the lifetime of a variable to be split; during a variable's lifetime it may reside in several different registers as well as in memory. Thus, exhaustively enumerating all the possibilities that Chow's algorithm might consider requires a two step approach. The first step is to construct graphs corresponding to all the possible ways that variable lifetimes can be split. The second step is to exhaustively color each of these graphs using the same exhaustive enumeration algorithm already described. A dynamic programming approach can be used to reduce the size of the search space. Once a solution is found, the enumeration only needs to consider potential solutions that require fewer saves and restores. For example, if the enumeration finds a solution that requires only two register saves and restores, it does not need to consider graphs that split three or more lifetimes.

The comparison of this enumeration method with the other allocation strategies is shown in Figure 4.4.

Method	% temp access in mem
Pure Chaitin	52%
Modified Chaitin	42%
Temp Enumeration	39%
Chow	61%
Full Enumeration	38%

Figure 4.4 – Utilization of registers for temporaries on the VAX with three registers available for allocation.

The Chow method did poorly for this type of register allocation as its greedy assignment algorithm resulted in picking nodes to place in registers that were often better left in memory. While the Chow method appears superior to any other method for the large problems of procedure wide register allocation, it does not handle the local problems as well as other heuristics. The full enumeration policy nearly always made the same selection as the faster enumeration policy that did not consider splitting lifetimes albeit much more slowly.

4.3.3. Constraints on Temporary Nodes

Machines such as the VAX have a uniform register set, and an orthogonal instruction set that allows operands to be interchangeably accessed from either memory or registers. Most machines do not have such flexible architectures. On machines with one and a half address instructions such as the IBM 370 or the MC68000 at least one operand must be in a register. Thus, certain temporaries are constrained to be allocated to registers. For

such machines a pass must be made over the tentative code generation to flag any temporary operands that must be allocated to a register. Flagging temporaries can be done as explained in section 3.3.2. A table constructed at code generator construction time can be consulted. Alternately, the code generator can be probed with the different types of potential temporaries to see whether a single instruction can be generated. Those uses of temporaries that must be allocated to registers can then be annotated to be restricted. For machines such as the MC68000, several tests must be done, such as the viability of using address versus data registers as well as a memory location.

Once the conflict graph has been constructed but before any of the coloring algorithms are run, the graph is partially colored to reflect the constraints. If there are so many constraints that the coloring is impossible then the lifetimes of the constrained nodes must be split. Either Chow's algorithm or the full enumeration algorithm can be used to do the split. Splitting of the constrained nodes continues until a coloring is possible. Once a coloring has been achieved for the restricted subset, any of the other heuristics can be used to finish the allocation.

Dealing with the issue of different types of registers such as address and data registers on the MC68000 is also straightforward. Most temporaries may be allocated in only one type of register (or memory). To do register allocation, the problem is divided into several subproblems. Dependency graphs are created that contain only the nodes that must be allocated to a particular type of register. Each of these graphs is independently colored using any of the coloring heuristics described above; the algorithm uses the pool of available registers of the required type. Finally a complete dependency graph is constructed and all the registers are combined into a complete pool. The colorings from the sub graphs are placed into the complete graph as initial dependencies and a final coloring is done in the usual way.

This method can do an optimal job of allocating the temporaries that are constrained to be of a particular type. It does tend to bias against those temporaries that can be in any type of register since they are given lower priority for allocation. In practice the number of such temporaries is small and can usually be accommodated.

4.4. Comparison with Other VAX Code Generators

The objective of the register allocation strategies is to minimize the number of static memory references, on the assumption that such a policy minimizes the number of dynamic memory references. To provide a comparison of the overall effectiveness of our register allocation strategies the typesetting test programs described in section 2.5 were run through the one and multi-pass (set to run in two pass mode) code generators and two other code generators available to us. The two others are the UNIX production code

generator [Lions79], and the more recent version, pcc2, that uses a dynamic programming approach [Henry81b]. The tests were run on both unoptimized IR and IR that had been run through Chow's optimizer to do procedure wide optimizations including register allocation. Because pcc2 accepts a different IR than the other code generators, it could not be tested with the optimized IR input.

The optimizer was modified so that the register directives given by the programmer could be either honored or ignored. If the directives were to be honored, Chow's optimizer allocated only those registers in the dedicated partition that were not already used by the programmer. If the directives were being ignored, Chow's optimizer allocated from among all the registers in the dedicated partition. In the experiments below the register directives were always honored. Unlike other optimizing compilers that we have measured, our compiler was able to ignore the register directives without changing the running time of the **troff** benchmark.

4.4.1. Benchmarks of the C Typesetting Program troff

Seven executable versions of the C typesetting utility **troff** were produced. Each target program was timed on how long it ran to typeset the nineteen page manual entry for the C-shell. The output of the **troff** compiled with the production compiler was saved to compare against the output of all the other runs to insure that they not only ran quickly, but also produced correct results. The results from these tests are shown in Figure 4.5. Times are in seconds, percentages are relative to unoptimized pcc.

code generator	without tree opt	with tree opt
pcc	143.3/100%	129.4/90%
pcc2	141.8/99%	N/A
GG one pass	141.6/99%	127.3/89%
GG multi pass	140.0/98%	121.3/85%

Figure 4.5 – Running time of the C type setting utility **troff** on the VAX 11/780.

From a macroscopic point of view the differences between the running time of the utility without the benefit of the tree optimizer is not statistically significant. This small difference stems from the low level nature of the C language. Most of the statements in the program have a single obvious translation that all the code generators find. The only differences are in the few complex expressions. These expressions do not occur in parts of the program that are frequently executed, hence their optimization does not significantly affect the running time of the program.

By contrast the information provided by the combination of the intermediate tree optimizer working with the code generators provides a measurable improvement. All the code generators improved their code generation about equally with the tree optimized input, since they all generated identical code for most of the (improved) intermediate expressions. As the next section shows, greater gains are possible for languages such as Pascal, since Pascal does not have unbound pointers that limit the optimization of C.

Most of the gains by the multi-pass code generator were derived from its use of excess scratch registers. The scratch registers were used to cover common subexpressions found but not assigned to a register by the tree optimizer. The reason that the multi-pass code generator had excess scratch registers is that the register partition on the VAX contains too many registers. Because the block move instruction can modify the first six registers, all these registers are considered scratch and hence are in the domain of the code generator. After deducting architecturally defined registers, only six registers remain for the tree optimizer to work with. The net effect is that the tree optimizer often has expressions that it would like to place in registers but it does not have enough registers to do so. By contrast, the code generator almost never needs all the scratch registers in its domain. Hence, the ability of the multi-pass code generator to allocate its temporary registers to the expressions found by the tree optimizer pays off significantly. Presumably the other code generators would be able to show similar improvements if they were modified to interact similarly with the tree optimizer.

Judging from the results of the MC68000 with only four registers in its scratch partition, if the VAX's partition were changed so that the code generator had only three or four registers, the gains would probably be less dramatic. The change could be handled by having the code generator save and restore registers r4 and r5 whenever it generated a block move instruction. The other possibility would be to have the IR optimizer know the IR nodes that could potentially generate a block move instruction and have it kill any values it had in r4 and r5. The latter solution would introduce undesirable machine dependencies into the IR optimizer.

4.4.2. Benchmarks of the Pascal Typesetting Program TeX

Six executable versions of the Pascal typesetting utility TeX were produced. Because there is no Pascal front end for pcc2, it could not be included in these benchmarks. In the examples below, "pcc" refers to the portable C compiler back end; the IR is always that produced by Chow's Pascal front end. Each target program was timed on how long it took to typeset a thirty-two page manuscript that contained many mathematical equations and special font requests. The output of the TeX compiled with the production compiler was saved to compare against the output of all the other runs.

Unfortunately, the output from the tree optimized runs failed to match the expected result. An inspection of the typeset output shows that the widths of special characters were being incorrectly computed. The problem appears to be caused by some interaction between Chow's Pascal front end and the IR optimizer. The unoptimized output from the Pascal front end works fine and input prepared from the UNIX C front end runs through the optimizer without problems. The code generators do not appear to be at fault since they all generate binaries that produce the same incorrect output. The results from these tests are shown in Figure 4.6.

code generator	without tree opt	with tree opt
pcc	233.1/100%	142.2/61%
GG one pass	207.7/89%	125.9/54%
GG multi pass	202.5/87%	114.2/49%

Figure 4.6 – Running time of the Pascal type setting utility TeX on the VAX 11/750.

The ten percent improvement in the running time of the single pass code generator without the benefit of the tree optimizer comes from its use of the indexed addressing mode on the VAX. The pcc code generator seldom manages to use the indexing addressing mode on the VAX. The capability to use indexing is more important for Pascal programs than it is for C programs because Pascal programmers do not have the option of using pointers to access data structures declared as arrays. Thus, the higher frequency of array accesses provides more opportunity to use the indexing mode on the VAX. As with the **troff** example, there are few complex expressions requiring the capabilities of the multi-pass code generator. So in the absence of the common subexpression information, the multi-pass code generator is not able to do much better than the simple one pass code generator.

In sharp contrast to the C program example, the tree optimizer nearly doubles the running speed of the program. Relieved of the burden of unbound pointers and programmer directed register allocations, the tree optimizer manages to find many opportunities for improvement during the seven hours of VAX 11/750 time that it spends optimizing the IR. As in the unoptimized case the one pass code generator manages to capitalize on the indexed addressing mode of the VAX to generate better code than the pcc code generator. With the addition of common subexpression information, the multi-pass code generator again manages to use the extra scratch registers on the VAX to good advantage to reduce the running time by an additional 5% over the single pass version.

4.5. Comparison with Other MC68000 Code Generators

A similar set of comparisons were run on the MC68000. In this experiment the production compiler is the one available on the Sun Microsystems workstation; like the VAX production compiler, Sun's compiler is derived from the original pcc. Unlike the production pcc on the VAX, the Sun compiler has been hand tuned to catch many special cases that the standard template driven version of pcc would not be able to find. For example, it converts multiplies and divides by a constant into a series of shifts and adds. So it produces better code than a normal retarget of pcc. Because the language specific front end of Sun's pcc is combined with the code generator into a single program, the tree optimized version example was obtained by running the source through the UNIX C front end, then through Chow's optimizer and finally through only the back end code generation of Sun's pcc. Consequently optimizations done by their front end are lost, making the comparisons less accurate than they would be if the Sun front end could be used. As before, pcc2's incompatible IR could not be run through the optimizer.

4.5.1. Benchmarks of the C Typesetting Program troff

The results of the **troff** experiment are shown in Figure 4.7.

code generator	without tree opt	with tree opt
Sun pcc	175.4/100%	164.9/94%
pcc2	176.1/100%	N/A
GG one pass	188.2/107%	171.4/98%
GG multi pass	185.6/106%	166.5/95%

Figure 4.7 – Running time of typesetting utility **troff** on the Sun 1.1 work station using a 12 Mhz MC68010.

The hand tuned pcc code generator does significantly better than either the single or the multi-pass code generator. With the aid of the tree optimizer our code generator produces code that runs slightly faster than pcc. However, Sun's pcc code generator gains a similar improvement from the optimized IR since its optimizations are disjoint from those done by the tree optimizer. The difference in timing is not caused by the register allocation; the coloring algorithm generates fewer spills than pcc's hand tuned algorithm. A reason for the smaller improvement of the multi-pass code generator on the MC68000 than on the VAX is the more reasonable register partition. There are ten registers available to the tree optimizer and only four dedicated to the code generator. This partitioning provides fewer opportunities for the code generator to apply excess registers to tree optimizer expressions; in addition the tree optimizer has fewer expressions that it is unable to place in registers.

4.5.2. Benchmarks of the Pascal Typesetting Program TeX

The results of the TeX experiment are shown in Figure 4.8. As in the previous TeX example, pcc2 is not included because of its lack of a Pascal front end. A comparison of the output failed in the same way as in the VAX example.

code generator	without tree opt	with tree opt
Sun pcc	168.9/100%	92.9/55%
GG one pass	172.3/102%	94.6/56%
GG multi pass	169.6/100%	87.8/52%

Figure 4.8 – Running time of typesetting utility TeX on the Sun 1.1 work station using a 12 Mhz MC68010.

All three code generators handle the unoptimized trees about equally well. The hand tuning of Sun's pcc does not have as much of an advantage over our code generators as it does in the C example. We do not have any explanation for the lesser improvement for this experiment. As with the VAX, the tree optimizer is able to nearly halve the running time of the program. The improvement of the multi-pass code generator is smaller than on the VAX because of the smaller number of registers available for holding common subexpressions.

4.6. Improvements Using Dynamic Programming

[Ripken77] introduced the idea of using dynamic programming techniques to systematically consider the numerous possible sequences that can compute a given expression tree. Unfortunately, the running time of this algorithm increases exponentially with the complexity of the machine architecture. Research in this area is continuing to find ways of using it in a limited way to derive its benefits without incurring an excessive time penalty [Christopher84].

This technique can also be extended to evaluate the costs of different spill choices; most operations in the tree for which code is being generated can be done in several different ways by the target machine. The set of alternative solutions appear as shift-reduce and reduce-reduce ambiguities in the machine description grammar that produces the code generator parser. Currently these ambiguities in the grammar are resolved statically by the program that generates the parser tables. For each conflict it decides which alternative is better in the most cases and creates a parser that will deterministically choose that alternative.

Usually static selection is reasonable; however, there are some notable exceptions. For example, on a one and a half address machine, either operand may be in memory but

not both. Consider the case where the parser is matching a node in the tree that has two memory operands. Under the currently used maximal munch static allocation policy, the parser will always generate an instruction that accesses the first operand from memory and require that the second operand be in a register.

One alternative to making all the parsing decisions at code generator construction time is to leave some of the decisions to code generation time. Specifically, a new alternative would be introduced to the parser that would be called "split". When one of these alternatives is encountered during parsing, the code generator would "fork" a clone of itself. For shift-reduce conflicts one clone would shift while the other reduced; for reduce-reduce conflicts the clones would reduce on the different possibilities. The progress of each of the parsers would be tracked using the techniques of dynamic programming. Each parse would maintain a cost associated with its instruction sequence based on the number of instructions generated, or if known, the number of machine cycles required to run the instruction sequence. As costs for a given parse exceeded known solutions they would be aborted.

The benefits of this technique would be that it would allow the code generator to consider alternatives that are resource dependent. For example, on the MC68000 there are many operations that can be done in either address or data registers. The choice of which type of temporary to use is dependent on which type of register is free. If the dynamic programming approach were available, then decisions on which type of temporary to use could be based on an entire expression rather than needing to be made based on a single instruction.

As an example consider the expression to round up to an even number:

$$i = (i + 1) \textbf{ and } (\textbf{not } 1)$$

On the MC68000 the first operation (addition) can be done in either an address or a data register. However, the **and** operation can only be done in a data register. The dynamic programming algorithm would consider both types of registers when generating the addition. Only when it continued up to the **and** would diverge and find that choosing a data register is a better solution.

The drawback to using dynamic programming is that it introduces much parallelism into an algorithm that previously had been linear. Since most of the static decisions that are made are always valid, some static pruning would probably insure that the amount of parallelism would be kept to a useful level. Pragmatically it would be useful to have a mode in which dynamic programming could be turned off so that the linear code generator could be used for the usual cases in which the simplicity of the input expressions precluded the need for the extra flexibility. This is an area for future research.

CHAPTER 5

Issues in the Design of an Intermediate Representation

This chapter reviews the general goals that an IR should fulfill. An overview of the topic can be obtained by reading from Chow's comprehensive bibliography [Chow83a]. The bibliography has been updated recently [Ottenstein84]. The second part of this chapter highlights the problem areas with the production IR that we use and suggests revisions to bring it closer to an ideal form.

5.1. Requirements of an IR Design

The design of the IR is important to insure flexibility and modularity in the resulting compilation system. It must contain enough information to allow the optimizer and code generator to produce good code; at the same time it must be kept simple. The cost of writing, storing, and reading the IR between compilation steps can cause a significant slowdown in the compilation system, particularly if the IR is unnecessarily overloaded with extraneous information. The most important goal is that the IR design should be as independent of the source languages and the target machines as possible. At the same time it should support as many languages and target machines as possible.

There are three major forms used as IRs. The highest level IRs are trees. One extreme example of trees is the Diana intermediate form used by Ada[1], that allows the original source text of the program to be recovered [Goos83], [DOD83]. Below the level of trees are postfix notations for stack machines. Unlike trees these notations fully specify the order of evaluation of all expressions. Finally at the lowest level are tuples or pseudo machine instructions. These forms typically try to model a class of target machines, identifying all intermediate results and addressing modes. An IR can be designed as either an *intersection* IR or a *union* IR [Lamb83]. A union IR tries to provide all the operations available in all the source language front ends. By contrast an intersection IR tries to provide just the set of primitives needed to evaluate expressions and generate control flow.

[1]Ada is a registered trademark of the US Government (Ada Joint Program Office).

For maximum retargetability, an intersection IR approach is best. The IR should fully specify all the language dependent operations, but should not constrain order of evaluation any more than necessary to conform to the semantics of the source language. The IR should not try to identify addressing modes; they should be fully exposed so that the code generator can find addressing modes according to the capabilities of the target hardware. The IR should contain nodes for all the basic operations supported by target processors. These operations include both the primitive arithmetic operations such as addition, subtraction, multiplication, division, and arithmetic shifting as well as the primitive logical operations such as **not**, **and**, **or**, and logical shifting. The IR should not include language specific operations. For example, Pascal set addition should be handled by an earlier phase that should either build expression trees to evaluate sets from the primitive operations or should insert library calls. Similarly the IR should not contain operators corresponding to fancy instructions on certain target machines. By selecting a small set of primitive operations, most target machines will be able to use a single instruction to execute the required operation.

Front ends can be tailored to use the fancy instructions on a particular target machine by passing the operation through as a subroutine call. The subroutine call can later be replaced in the assembly language output by the code generator with the special instruction or sequence of instructions to do the operation [McKusick83]. Alternately, the code generator constructor can analyze the complex instructions and determine where they can be used [Morgan82a], [Morgan82b].

Regardless of how the final instructions are generated, the front end should not be involved in their selection. The most important reason for having the front end pass through the operation as a subroutine call is retargetability. The front end should contain as few machine dependencies as possible. Emission of instructions requires knowledge of the instructions involved, and knowledge of how to access its operands. In addition, the subroutine interface decreases the amount of work required to retarget the front end. The initial retarget can simply use the slower library routines that can be easily ported since they are written in a high level language. Once the front end has been retargeted, the implementation can be speeded up as time and ambition permit with inline expansion or through the aid of an instruction analyzer.

The IR must explicitly enumerate the operations to be done; the control flow in the program should be fully defined. For example, the IR should not contain short circuit operators. Rather, the precise evaluation order should be specified with conditional branches used to exit the evaluation at the appropriate point. Requiring the front end to do full enumeration of control flow is reasonable since the transformation from short circuit to explicit evaluation is machine independent. However, the transformation from

short circuit to explicit evaluation is also language independent so it seems unnecessary to have every front end include the code for doing the transformation. One alternative is to add a third pass responsible for doing language and machine independent transformations such as converting short circuit evaluation into explicit control flow. Pragmatically a third pass is unnecessary since these transformations do not require any forward context; they can be done as part of the compiler system supplied output routines from the front end or as part of the compiler system supplied input routines to the code generator.

An important property of IR design is that the IR contain complete information about the variables that it references. The IR optimizer needs to be able to distinguish variables that are in memory from those that are in registers. These distinctions conflict with the goal of keeping the IR machine independent since a complete specification of variable access is inherently machine specific. To maintain machine independence, the variables must be referred to symbolically in the IR. The front end must provide a simple symbol table for the code generator to use to allocate and assign addresses to variables [Kornerup80]. There are two major flaws with this approach. Pragmatically we want the IR optimizer to be able to recognize the location of operands and to be able to change memory based operands to be in registers. For operand recognition to be possible, the bindings of variables to addresses must have been done before the code generator is run. From a modularity point of view, all the symbol table binding should be done by the front end. The next section discusses how to provide a target machine independent interface for doing variable allocation in the first pass.

The IR design must allow the front end to provide information that enables the IR optimizer to detect potential aliases so that it can safely calculate common subexpressions and shadow the values of variables in registers. Specifically, the front end must specify the set of objects that may be modified by any assignment through addresses computed at run time. The simplest case is for array references; the information specifies the lower and upper bounds of the array as the range of addresses that may be modified. More difficult are programmer defined pointers. In Pascal pointers are constrained to point to the heap; in C they are completely unconstrained. Parameters passed by address also introduce many potential aliases. In the worst of these cases the front end may simply have to specify that potentially they point to everything. One important fact that the front end can typically discover is whether the pointer can be excluded from pointing to any objects in the frame of the local routine. Since most accesses are to local variables, knowing that the pointer does not access any local variables reduces the amount of information that the optimizer must discard.

The other main feature that is added to the IR design primarily for purposes of optimization is the ability to distinguish between variables and temporaries. Often the

first pass needs to allocate a temporary to hold a common subexpression or an intermediate result. Because these temporaries are typically allocated in the local stack frame, they are indistinguishable from local variables. If these temporaries are distinguished by having their own IR node then the optimizer can do special optimizations such as discarding the memory assignments if the value can be used directly from the register in which it is computed.

The approach that we use is to introduce two new nodes, ASSIGN_TEMP and USE_TEMP. The ASSIGN_TEMP node is a binary operator whose value defines the name of the temporary, its left child is a memory location in which it can be saved, and its right child is an expression representing the value of the temporary. The USE_TEMP node is a leaf node whose value defines the name of the temporary to be used. These nodes allow the IR optimizer or code generator to decide whether to replace all the uses of the node with the tree corresponding to the temporary or to compute and save the subexpression. If the cost of computation of the temporary is sufficiently high to warrant computing and saving its value, then the decision of whether to place it in its default memory location or in a register can be made based on available register resources.

These nodes have been added to our IR. None of the front ends currently generate them, however, the IR optimizer uses them to identify common subexpressions that it has not been able to put into registers. The multipass code generator can then use any excess registers in its partition to hold the subexpressions. Often the common subexpression represents a single addressing mode on the target machine, so it is not worthwhile to precompute it unless a register is available. When a register is not available, the code generator simply expands the temporary rather than computing it to some other memory location.

One important issue that arises with these subexpressions is whether they have side effects. The semantics of the temporary node are that it may be evaluated one or more times. Thus, the front end and optimizer must insure that they do not use the temporary nodes for expressions with potential side effects. The IR optimizer typically must be more conservative than the front end. For example, the front end may place a function call as a common subexpression if it can determine that the function call has no side effects or that the language does not preclude multiple evaluations of expressions. By contrast the optimizer must assume that function calls always have side effects and avoid the possibility of multiple evaluation.

5.2. Problems Encountered with a Production IR

The IR accepted by our code generator was chosen because we had compilers that used it rather than for its design. It was informally created by two people that were

building an interface between a C language front end and a code generator for a specific machine [Johnson78]. The IR was further evolved to accommodate a Fortran front end [Feldman79]. The effect of this lack of design is that the IR has several operators that are specific to a source language and also constructs that are machine dependent. The remainder of this chapter explains the problems that we encountered and gives proposed solutions to resolve the problems in the current design.

5.2.1. Machine Dependencies

The language has numerous examples of unnecessary machine dependencies that appear explicitly in the IR. The fundamental problem is that the IR allows assembly language for the target machine to be passed through the code generator. The first problem with having assembly language in the IR is that the code generator must either generate assembly language that can be subsequently passed through an assembler or it must contain an assembler so that it can translate the assembly language to machine code. While there are good reasons for having an assembler run after the code generator to handle such problems as span dependent jumps and forward references, the code generator must still be able to interpret the assembly language if it is to detect the beginning and end of blocks caused by labels and branches.

This problem is acute in the IR that we use since it lacks several important control flow nodes. Specifically, labels and unconditional branches are passed through from the front end as assembly language strings. Conditional control flow is passed through as IR nodes with the target of the conditional branch specified as an integer value in the node. The target label however, is still put out as an explicit assembly language string; the front end must know the algorithm that the code generator uses to convert an integer into a label string. These problems can easily be alleviated by simply providing nodes in the IR for unconditional jumps and label nodes.

Another missing control flow node is one for multi-way branches; they must be emitted as assembly language by the front end. The front end emits the case expression as an IR tree, then emits a node that requests that the result of the expression be pushed into a known register. It must then emit a set of assembly language instructions to compute the jump using either a series of if-then-else constructs for small or sparse sets of target labels, or a computed jump and jump table for large or dense sets of target labels. Thus, every front end has to duplicate the analysis of how to build the case instruction, and must also include often complex sequences of instructions on each of its potential target machines to generate multi-way branch instructions.

Again the solution is simple. The IR should have a binary MBRANCH node that takes as its left argument an expression and as its right argument a list of value-label

pairs corresponding to each of the targets. The decision on whether to implement the multi-way branch as a series of if-then-else constructs or as a jump table is target machine independent. The language and machine independent pass could choose to rewrite those MBRANCH nodes not suited to implementation with jump tables with if-then-else constructs. Thus, the code generator could always generate jump tables for the MBRANCH nodes that remained.

Another problem related to control flow is the code associated with routine entry and exit. Although the code at the point of call is handled entirely by the IR, the code for procedure entry and exit is passed through as assembly language by the front end. Thus, the knowledge of the calling convention is split between the front end and the code generator. This split makes it more difficult to experiment with different calling protocols since each new protocol must be changed in all the front ends and code generators.

The proposed change is to add IR nodes for routine entry, exit, and exit with return value. Unfortunately, routine entry is not entirely language independent. Scoped languages such as Pascal may require additional entry and exit code to update the display. One alternative is to provide entry and exit nodes for each language that the code generator supports. We rule out this possibility as it violates the principle of keeping the IR language independent. Another alternative is to generalize the entry and exit nodes to cover the requirements of all the languages. This proposal has two drawbacks, first it penalizes simple languages such as C that need only a minimal interface; second another language may come along with requirements that the general node does not handle. At a minimum changing the entry and exit protocols requires recompiling all the libraries, and completely recompiling any programs that are changed in any module (so that they can be relinked to the new libraries).

The alternative that we propose is to provide the simplest entry and exit protocol that can handle argument passing and allocation of local variables and temporaries. More complex protocols, such as displays or static links, are managed with additional IR output by the front end. Thus, to set up a static link, the Pascal front end allocates a compiler generated local variable in the stack frame or a TEMP node and generates an IR tree to assign the static link value to it. Similarly, immediately preceding routine exit, the front end can emit IR trees to restore display values or do other language specific bookkeeping before the routine exits.

Another benefit of having the code generator handle routine entry is that it can also be responsible for generating calls on the profiling system rather than having the front ends generate explicit assembly language to do so. To retain the current flexibility allowing selective profiling of routines within a module, a new IR node that requested entry and profiling code could be added in addition to the simple entry only IR node. In

practice this flexibility is never used, so an alternative would be simply to pass the profiling request to the code generator instead of the front end and have the code generator augment all entry requests with a profiling call.

Currently the mechanism for returning values from routines is known by the front ends. Although scalar return values are straightforward, structured return values are complex. Some of the front ends directly generate assembly language; others produce IR trees to do the necessary assignments. However, the protocol is hard wired in all cases. A better solution is to provide IR nodes that differentiate between function calls (that return values) and procedure calls (that do not return values). The function calls specify the size of the quantity that they are returning so that the code generator knows whether it can use the (usually) faster scalar return protocol or the more general structure return protocol. The IR node for return is no longer constrained to be a leaf node, but becomes a unary operator that takes an expression to return as its child (procedures specify a null child). The front ends are no longer dependent on the mechanism used to implement the function return protocol.

5.2.2. Machine Versus Language Issues of Addressing

The issue of where and how to do variable allocation is the most difficult issue to resolve in IR design. The decisions on what data types to use and where the variable is allocated (global or local) are language dependent issues. To maximize the exposure of the operations to the optimizer, the precise location and address calculation are desirable so that the optimizer can make informed decisions about how best to minimize expensive accesses. Conversely, the size used to represent the data type, and its precise location and alignment in its allocation area are machine dependent issues. The present IR accomplishes the first two points well as the front end is responsible for variable allocation down to the level of deciding offsets from the local frame pointer. Unfortunately, this level of specification introduces machine dependencies into the front end.

An ideal scenario might be to set up a connection between the front end and the code generator for the target machine. The initial negotiation would be over how to represent the various data types in the language being compiled. Consider a Pascal program that needs to represent a subrange type that is declared as:

type tag = 0..240;

The Pascal front end would query the code generator with the subrange and the code generator might repond that the appropriate type is "unsigned char" for a byte addressed machine, while it might respond "word" for a word addressed machine. Similarly the Pascal front end would establish a machine type for the basic language types such as "integer", using the minimum precision that the language specification of Pascal allows.

Once the fundamental types had been set up more complex data structures could be constructed. Consider the Pascal record declared as:

```
type unit = record
        unittype :tag;
        value :integer;
end;
```

The front end would inform the code generator that it was starting a new record. It would then specify the types of each of element in turn, getting back appropriate offsets to access them.

The allocation of variables would proceed in a similar manner. The type of the variable and its location (local or global) would be passed to the code generator. The code generator would return an appropriate IR tree for accessing the variable that would be stored as part of the information associated with the variable.

This scenario has the benefit that the IR does not lose any of the information that it currently contains. All the machine dependent decisions such as where variables should reside, what alignment they should use, and how much space they need are handled by the code generator. The problem with this scenario is that the code generator and the front end must run as co-processes, set up possibly expensive communication channels, and reestablish the basic machine types on every run of the compiler.

One alternative is to have the front end preface each routine with a list of type requirements and the list of variables in a simple symbol table. It would then refer to the variables symbolically in the expression trees. The code generator would then resolve all the symbols in the expression trees just as it would have done interactively. The problem here is that the code generator now has to deal with a symbol table, a task that should be in the domain of the front end. Also the IR optimizer now has less information unless the resolving part of the code generator is run first. Finally the high dynamic cost of recalculating the basic type alignment must be paid on every compilation. On the positive side the IR is completely machine independent.

To mitigate these problems, the resolving part of the code generator is broken out into a separate module that answers its questions by consulting a table. The code to query the table is compiled into each front end. As part of its startup, the front end loads the table for the appropriate target machine, thus allowing it to emit fully specified IR. To eliminate the per run costs of establishing the basic types of the language, the front end creates its own initialized tables by reading in the pure machine tables, running its basic language type queries, then writing out the tables.

In summary, the front end should do all the symbol table resolution. This decision introduces some of the machine dependent choices into the IR trees, however, these

dependencies are useful to the IR optimizer. Unlike the current scheme that compiles the machine dependencies into the code, the machine specific information is dynamically loaded and described in a modular table driven way.

5.2.3. General Poor Design Choices

There are several problems with our production IR that are caused by its initial use for the C language and the Digital Equipment Corporation hardware. The most blatant example of its heritage is in a special set of IR nodes to handle the side effect operators post-increment and pre-decrement. These operators are designed to emulate the automatic post increment of registers found on some hardware. These operators violate several design criteria. First they implement a special feature of the C language that can be constructed from simpler operators. Second they are not directly implemented on many target machines. They must be implemented either by rewriting the IR to use simpler operations, or by constructing a set of instructions to do the same thing. Finally they have side effects; if one of these nodes is duplicated, the operation is incorrectly done an extra time. The number of side effect nodes in the IR should be kept to a minimum to simplify the task of finding side effect nodes (optimally only routine calls) thus maximizing the opportunities for using temporary nodes.

Another shortcoming is the relaxed requirement in the IR for explicit type conversions between scalar data types. Because of the way that the VAX accesses memory, no conversions are required between character, short, and long data types. The IR should require that all conversions be explicitly specified. Those target machines that do not need to do any conversions can simply ignore the conversion operators.

The IR contains FIELD operators for inserting and extracting bits from larger addressable units on the target machine. The FIELD operators can be composed from simpler shifting and masking operands, hence they could be dropped from the IR without any loss of functionality. Some machines have field insertion and extraction instructions; these instructions can easily be generated by passing through a subroutine call that can be replaced by the inline code expander with the appropriate special instruction. One argument against this solution is that enough machines provide sub-word addressing in their hardware that the node should be retained. For these machines the inline expansion will result in a load instruction where the operand could previously been used directly in a larger operation. To avoid the use of the load instruction, the code generator would have to be able to recognize the field addressing mode from the primitive IR nodes. If the field node is left in the IR specification, it should be redesigned so that its alignment and width are specified as children rather than being encoded in its value attribute.

The IR includes a set of nodes designed to implement a conditional expression that appears in the C language. This node is an historical artifact; the first pass should expand this control structure into basic expressions and conditional branches just as it does for all the other control structures in the language.

Finally there are several minor problems noted here for completeness. The node for structure assignment is inconsistently missing a level of indirection; unlike all other assignment nodes it requires its right hand child to be an address rather than a value. Similarly the node for doing routine calls takes its entry name as an address rather than as a value. Another minor problem with the IR is that it distinguishes between routine calls with and without arguments. The distinction is unnecessary since a call without arguments can be trivially determined by noting that the node has no right child. Finally the IR does not distinguish between procedure and function calls. As stated above this distinction is necessary to allow the value return to be handled by the code generator rather than being known to the front end.

CHAPTER 6

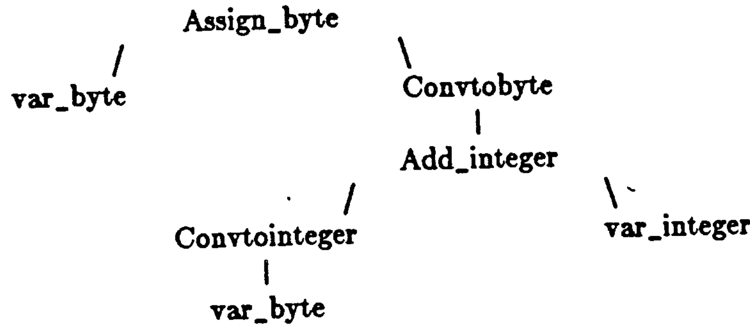
Data Conversion in Machine Independent Code Generators

A major task performed by modern programming languages is to map the many hundreds of abstract types defined and used by a large program into the small set (typically one to ten) of predefined types available in the base language [Leavens84]. These predefined types must in turn be mapped into the data types available on the target machine. The target machine may be a simple RISC machine providing a single scalar data type from which all other types must be derived. Conversely, the target machine may be more complicated. An example is the VAX, that provides four scalar formats, four real formats, and a decimal format. Usually the high level language allows the different types to be intermixed in a single expression. For example, an expression may contain both scalar and real numbers. The type mapping and the evaluation of mixed mode expressions all require data conversion.

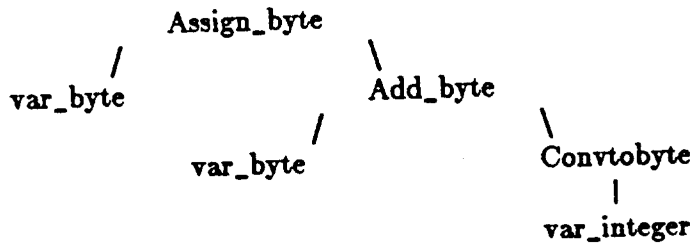
6.1. Data Conversion Issues

Data conversion requires decisions to be made throughout the compilation process. The first set of issues are the language dependent issues that must be handled by the front end. The language specifies conversions that must be requested explicitly by the programmer, usually conversions that may lose information such as converting a real number to an integer. It also specifies those conversions that may be done implicitly, such as allowing arithmetic combinations of integers and reals. The language definition or compiler convention must specify what precision is used for intermediate results. For example, the language may require all integer operations to be done with 32-bits of precision.

The decisions that are made by the earlier phases are shown explicitly in the IR. All operators must have their type specified; the children of each operator must either match the type of the operator or contain a conversion from their type to the type of the operator. For example, if a byte and an integer are being added together and assigned to a byte there are at least two choices of how to do the computation. The first is to use full integer precision producing a tree such as:



In this example the use of the var_byte must contain a conversion of byte to integer and the sum must be converted from integer to byte before being assigned to the var_byte. The second choice is to compute the expression using byte precision producing a tree such as:



In this example the var_integer must be converted from integer to byte before being added to the var_byte. The choice of which variant to use depends on the semantics of the programming language.

The main point of contention is whether the types shown on the nodes should represent the base types of the language or the target machine types. As discussed in chapter 5, we believe the IR should reflect the native types available on the target machine. This decision requires that the code generator pass a small amount of information to the front end describing the types available on the target machine. Using this information, the front end can fully resolve the language types to the target machine types. Note that the target machine information is not needed to determine where conversions are necessary. However, the machine information may eliminate the need for conversions between language data types that are mapped to the same native machine type.

The remaining data conversion issues are dependent on the target machine. They involve selecting code to do the conversions and allocating the necessary resources to hold the intermediate results. Generating code in the presence of mixed data type calculations can considerably increase the complexity of doing resource allocation over the simpler allocation strategies required when all calculations involve a single data type. For

example, consider an expression that contains both scalar and real subexpressions that are combined to yield a real result. Suppose the target machine has a single set of general purpose registers that hold either a scalar quantity or are paired to hold a real value. One way to minimize register usage may be to compute the scalar value first, but not convert it to a real value until just before it is used, since the subexpression uses a single register as a scalar rather than two registers as a real. Conversely, if the machine has separate real and scalar registers, it may be optimal to convert the scalar value before it is needed thus freeing up scalar registers for other calculations. The remainder of this chapter explains how data conversions can be handled in a Graham-Glanville code generator and motivates our decision on how to implement them.

6.2. Conversion Handling in Graham-Glanville Code Generators

All the analytic tools we have developed to produce Graham-Glanville code generators have been syntactically based. The reason that we have chosen this approach is to try to solve as many of the problems as we can with a single proven mechanism. Thus, if type conversion can be described syntactically, the existing tools will insure that the conversions can be done without causing the code generator to block and that they will always be done correctly.

Because typical target machines have only a few data types, it is feasible to describe conversions syntactically. For each operator and each data type a new terminal is created representing their combination; these terminals are then used in the machine grammar. For example, if the machine has a scalar and a real data type and an addition operator for both data types, the grammar would be expanded from having a single addition operator to having two addition operators, "Add_scalar" and "Add_real". The operands are similarly typed; thus instead of having simply "Memory" operands, there are "Memory_scalar" and "Memory_real".

Once the grammar is fully type specified, conversions can be specified as productions in the grammar. Thus, a conversion from scalar to real is specified as:

Temp_real ::= Convtoreal Temp_scalar

Associated with the production is the emission of the appropriate conversion instruction(s).

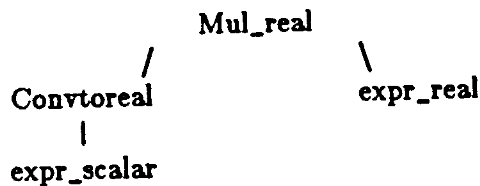
One feature that syntactic types introduces is that type conversions do not need to be placed explicitly into the IR. The machine grammar can allow implicit (presumably safe) conversions to make operands match their operators. Thus, if one operand of an integer add is a byte, a production of the form

Temp_integer ::= Temp_byte

will cause the byte to be widened to an integer before the addition is done. This feature is of dubious value since the front end should be aware of what data types it is using and emit the appropriate conversion operators rather than depending on the code generator to do the right conversions. Also the conversion operators must be present to allow the tree optimizer to minimize the register needs as described in the next section.

6.3. Optimizing Resource Utilization

A challenge in using a deterministic table driven Graham-Glanville code generator is figuring out how to prevent its left to right bias from generating premature conversions. The problem is best illustrated with an example shown in the following expression tree fragment:



Consider a general register machine that has five registers available for evaluation and uses pairs of registers for reals. Assume that "expr_scalar" requires four registers to evaluate and that the "expr_real" requires two register pairs to evaluate. The expression above can be evaluated without spills if the evaluation proceeds as follows:

1) compute the scalar expression

2) compute the real expression

3) convert the scalar to real

4) do the real multiplication

Because the parser walks the tree in a left prefix order, the left children are evaluated before the right ones. Because the last step in evaluating the left hand side is a conversion, the conversion will be done before the right hand side is evaluated, leaving too few registers to evaluate the right hand side. The only way to delay the conversion is to reorder the children of the real multiplication. However, the reordering leaves too few registers to evaluate the scalar.

The way that we have chosen to resolve this problem is to the modify the Sethi-Ullman numbering pass. As the algorithm moves up the tree calculating register needs it notes when it finds a conversion operator that widens its operand into one or more additional registers. If the extra register consumed by the conversion would exceed the number of available registers, the tree below the conversion operator is pruned off and

assigned to a common subexpression temporary. The common subexpression temporary is then put in its place below the conversion operator and the expression is flipped to put the conversion operator as the right child. This ordering forces the code generator to evaluate the scalar first, then the real operand, and finally the conversion as desired. The coloring mechanism described in Chapter 4 is then able to assign the temporary to a register.

6.4. Using Semantics to Handle Type Conversion

The benefits of distinguishing types syntactically are for the implicit conversions that they supply. Given the philosophical and pragmatic reasons for having all the type conversions explicitly in the IR, distinguishing types of operators and operands syntactically is less necessary. Thus, we have considered eliminating the syntactic distinction of types. All the blocking and correctness still apply to the explicit conversions, even if the types are not distinguished. The primary loss would be that type mismatches that had previously been handled by implicit rules or caught as an error would no longer be detected. These infractions could presumably still be caught through a semantic check of the type fields in the IR nodes.

The primary benefit of eliminating the syntactic distinction would be a reduction in the size of the machine description grammar. The magnitude of the decrease varies based on the number of data types supported by the machine and by how orthogonal its architecture. The number of productions in the grammar with and without syntactic types is shown in Figure 6.1.

Machine	with types	without types
VAX	858	440
MC68000	534	305
RISC	290	197

Figure 6.1 – Grammar sizes for various machines with and without syntactic types.

The number of data types has a direct correlation with the size of the grammar since each new data type adds productions for all the basic arithmetic operations. Thus, the size of the untyped VAX description is about half of the fully typed description because its set of nine types are reduced to one. The RISC has a smaller grammar to begin with, but does not decrease as much since it has only two data types. The MC68000 falls between the other two machines.

The more orthogonal a machine is, the greater the savings from eliminating syntactic types. The reason is that non-orthogonal machines must still differentiate certain operators by types. For example, the MC68000 must distinguish scalar addition operators

from real addition operators since scalar addition can be implemented by an instruction while real addition must be implemented by a subroutine call. Thus, the MC68000 loses proportionally fewer productions than the VAX because it is a significantly less orthogonal machine.

The savings is more dramatic when the size of the code generation tables is considered. Because the size of the tables grows as about the square of the size of the grammar, reducing the size of the grammar by half reduces the tables by a factor of 75%. We have not yet gathered any statistics on the change in running time of using typeless grammars since the front ends do not yet explicitly put in all type conversions. An alternative would be to have the tree rewrites add any necessary type conversions implied in the IR. Research is continuing in this area.

CHAPTER 7

Summary and Conclusions

Our goal has been to show how to integrate register allocation into a set of tools built in the context of a Graham-Glanville table driven code generator. These table driven tools allow the construction of high quality code generators for Von Neumann architecture computers in a short time and with a minimum of machine specific coding.

Previous researchers have tried to solve the register allocation problem with a single global register allocator that runs after code generation. We have taken a modular approach in which global register assignments are handled by a language and target machine independent optimizer and local register assignments are handled by the code generator. In addition to simplifying the code generator, this modular approach avoids the problem of over optimizing local register use at the expense of more global optimality.

The registers in our model are divided into a dedicated and a temporary partition. The dedicated registers are allocated by the language and machine independent optimizer that is run before the code generator. The optimizer is responsible for generating all spills and loads of the dedicated registers including those that must be saved across a subroutine call. The temporary registers are not saved across subroutine calls and are allocated by the code generator. The temporary registers are used to hold partial results as they are computed during the process of evaluating an expression. They are also used to hold the values of common subexpressions identified but not placed in registers by the optimizer and to hold intermediate results generated by low level rewrites of the IR trees.

Two register allocation models are described. They are driven by a description of the number and types of various registers and some policy description. The first is a one pass model that runs concurrently with the code generator and cannot take advantage of the hints provided by the optimizer. The one pass model does better than one would expect given its naive algorithms. The crux of the success is that most expressions are simple enough that the register manager never has to spill any registers. When spilling does occur, the furthest future use heuristic employed by the register manager works well.

The second model runs in multiple passes and tries to take maximum advantage of all the available information. It uses graph coloring to coalesce the common subexpressions found but not assigned to a register by the optimizer with the allocation of

registers needed during the process of expression evaluation. The solutions found by the coloring heuristics are shown to be nearly as good as the optimal solutions found by exhaustive enumeration. Two large heavily used typesetting programs are used to compare the effectiveness of the multipass register allocation strategies with the strategies used by the production UNIX compiler. The running time of both programs was improved by the register coloring strategies.

Both register models are flexible enough to describe several different register based machines. These machines include two reduced instruction set machines that have register windows, the complex but orthogonal VAX instruction set that has a uniform set of general purpose registers, and the non-orthogonal MC68000 that has two functionally overlapping sets of registers.

The main experimental result of this research has been that even when powerful register allocation techniques are used, program running time can only be modestly improved. In a language with uncontrolled pointers such as C, experiments showed a small running time improvement (1-2%) requiring a modest compilation time overhead (20%). In a language with tightly controlled pointers such as Pascal, experiments showed a modest running time improvement (10%) requiring about the same compilation time overhead (20%).

We have shown how to evolve the standard UNIX intermediate representation that binds the language specific front ends to the target machine dependent code generators so that it is both flexible and compact, thus insuring modularity in the resulting compilation system. The IR contains enough information to allow the optimizer and code generator to produce good code; at the same time it has been kept simple. Because the cost of writing, storing, and reading the IR between compilation steps can cause a significant slowdown in the compilation system, the IR has been stripped of any extraneous information. The most important goal of the IR design was that it be as independent of the source languages and the target machines as possible. At the same time it supports as many languages and target machines as possible.

Two semantic alternatives to the current syntactic specification of data conversions done by the code generator are described. One alternative is to require the front ends to explicitly include all necessary type conversions. The other alternative is to have the IR rewrites add any necessary type conversions implied in the IR from the front end. In either case the syntactic specification is no longer necessary in the machine grammar. Because the size of the tables grows as about the square of the size of the grammar, reducing the size of the grammar by half reduces the tables by a factor of 75%. Eliminating syntactic types also improves the running time of the code generator by reducing the number of chain reductions that serve only to track data types.

References

[Aho77]
Aho, J. D., and Ullman, J. D., "Principles of Compiler Design", Addison Wesley, Reading MA. 1977.

[Aigrain84]
Aigrain, P., Graham, S. L., Henry, R. R., McKusick, M. K., and Pelegri-Llopart, E., "Experience with a Graham-Glanville Style Code Generator", Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices, 19, 6, Montreal, Quebec. pp. 13-24. June 1984.

[ANSI77]
American National Standards Institute, "American National Standard FORTRAN", Standard X3.9-1978 (Fortran 77), American National Standards Institute, New York, NY. 1978.

[Barbacci77]
Barbacci, M. R., Barnes, G. E., Cattell, R. G., and Siewiorek, D. P., "The ISPS Computer Description Language", Technical Report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., August 1977.

[Berenbaum82]
Berenbaum, A., Condry, M., and Lu, P., "The Operating System and Language Support Features for the BELLMAC-32", Proceedings of the ACM Symposium on Architectural Support for Programming Languages and Operating Systems, SIGPLAN Notices 17, 4. Boston, MA. pp. 30-38. March 1982.

[Cattell77]
Cattell, R. G., "A Survey and Critique of Some Models of Code Generation", Technical Report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., 1977.

[Cattell78a]
Cattell, R. G., "Formalization and Automatic Derivation of Code Generators", PhD Dissertation, Technical Report 78-115, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., 1978.

[Cattell78b]
Cattell, R. G., "Using Machine Descriptions for Automatic Generation of Code Generators", Proceedings of the Third Jerusalem Conference on Information Technology, Jerusalem, Israel. pp 503-508. August 1978.

[Cattell79]
Cattell, R. G., Newcomer, and Leverett, B. W., "Code Generation in a Machine Independent Compiler", Proceedings of the ACM SIGPLAN 1979 Symposium on Compiler Construction, SIGPLAN Notices 14, 8. Denver, CO. pp. 65-75. August 1979.

[Cattell80]
Cattell, R. G., "Automatic Derivation of Code Generation from Machine Descriptions", ACM Transactions on Programming Languages and Systems 2, 2. pp. 173-190. April 1980.

[Chaitin82]
Chaitin, G. J., "Register Allocation and Spilling via Graph Coloring", Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices 17, 6, Boston, MA. pp. 98-105. June 1982.

[Chow83a]
Chow, F. C., and Ganapathi, M., "Intermediate Languages in Compiler Construction - A Bibliography", ACM SIGPLAN Notices 18, 11. pp. 21-23. November 1983.

[Chow83b]
Chow, F. C., "A Portable Machine-Independent Global Optimizer - Design and Measurements", PhD Dissertation, Technical Note #83-254, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305-2192, December 1983.

[Christopher84]
Christopher, T. W., Hatcher, P. J., and Kukuk, R. C., "Using Dynamic Programming to Generate Optimized Code in a Graham Glanville Style Code Generator", Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices 19, 6, Montreal, Quebec. pp. 25-36. June 1984.

[Cooper84]
Cooper, K. D., and Kennedy, K., "Efficient Computation of Flow Insensitive Interprocedural Summary Information", Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices 19, 6, Montreal, Quebec. pp. 247-258. June 1984.

[Digital77]
Digital Equipment Corporation, "VAX Architecture Handbook", Digital Equipment Corporation, Maynard, MA 01754. (617)-897-5111 1977, 1979, 1981, 1983.

[Ditzel82]
Ditzel, D., and McLellan, R., "Register Allocation for Free: The C Machine Stack Cache", Proceedings of the ACM Symposium on Architectural Support for Programming Languages and Operating Systems, SIGPLAN Notices 17, 4. Boston, MA. pp. 48-56. March 1982.

[DOD83]
Department of Defense, "Reference Manual for the Ada Programming Language", ANSI/MIL-STD-1815A-1983, American National Standards Institute, Washington, DC. January 1983.

[Ershov58]
Ershov, A. P., "On Programming of Arithmetic Operations", CACM 1, 8. pp. 3-6. August 1958.

[Feldman79]
Feldman, S. I., "Implementation of a Portable Fortran 77 Compiler Using Modern Tools", Proceedings of the ACM SIGPLAN 1979 Symposium on Compiler Construction, SIGPLAN Notices 14, 8. Denver, CO. pp. 98-106. August 1979.

[Ganapathi80]
Ganapathi, M., "Retargetable Code Generation and Optimization Using Attribute Grammars", PhD Dissertation, Technical Report #406, Computer Science Department, University of Wisconsin, Madison, WI., 1980.

[Ganapathi81]
Ganapathi, M., and Fischer, C. N., "Bibliography on Automated Retargetable Code Generation", SIGPLAN Notices 16, 10. pp. 9-12. October 1981.

[Ganapathi82a]
Ganapathi, M., and Fischer, C. N., "Description-Driven Code Generation Using Attribute Grammers", Conference Record of the Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, NM. pp. 108-119. January 1982.

[Ganapathi82b]
Ganapathi, M., Fischer, C. N., and Hennessy, J. L., "Retargetable Compiler Code Generation", ACM Computing Surveys 14, 4. pp. 573-592. Dec 1982.

[Glanville77]
Glanville, R. S., "A Machine Independent Algorithm for Code Generation and Its Use in Retargetable Compilers", PhD Dissertation, CS-78-01, Computer Science Division, EECS, University of California, Berkeley, CA 94720. December 1977.

[Glanville78]
Glanville, R. S. and Graham, S. L., "A New Method for Compiler Code Generation", Conference Record of the Fifth ACM Symposium on Principles of Programming Languages, Tucson, AZ. January 1978.

[Goos83]
Goos, G., Wulf, W., Evans, E., and Butler, K., "Diana Reference Manual, Revision 3", Tartan Laboratories, Inc., Pittsburgh, PA. February 1983.

[Graham78]
Graham, S. L., and Glanville, R. S., "The Use of a Machine Description for Compiler Code Generation", Proceedings of the Third Jerusalem Conference on Information Technology, Jerusalem, Israel. pp. 508-513. August 1978.

[Graham80]
Graham, S. L., "Table-driven Code Generation", IEEE Computer 13, 8. pp. 25-33. August 1980.

[Graham82]
Graham, S. L., Henry, R. R., and Schulman, R. A., "An Experiment in Table Driven Code Generation", Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices 17, 6, Boston, MA. pp. 32-43. June 1982.

[Graham84]
Graham, S. L., and Henry, R. R., "Machine Descriptions for Compiler Code Generation", Proceedings of the Fourth Jerusalem Conference on Information Technology, Jerusalem, Israel. June 1984.

[Henry81a]
Henry, R. R., "Building Compilers with Pcc2", Unpublished Paper, Bell Laboratories, Murray Hill, NJ. September 1981.

[Henry81b]
Henry, R. R., "The Code Generator's Work Station: Experiments with the Graham Glanville Machine Independent Code Algorithms for Code Generation", Master's Project Report, Technical Report M81/47, Computer Science Division, EECS, University of California, Berkeley, CA 94720. June 1981.

[Henry82]
Henry, R. R., "A Comparison of Three Different VAX C Compilers", Personal Communication, March 1982.

[Henry84]
Henry, R. R., "Graham Glanville Code Generators", PhD Dissertation, Computer Science Division, EECS, University of California, Berkeley, CA 94720. May 1984.

[Johnson77]
Johnson, S. C., "A Tour Through the Portable C Compiler", Bell Laboratories, Murray Hill, NJ. 1977.

[Johnson78]
Johnson, S. C., "A Portable Compiler: Theory and Practice", Conference Record of the Fifth ACM Symposium on Principles of Programming Languages, Tucson, AZ. pp. 97-104. January 1978.

[Joy79]
Joy, W. N., Graham, S. L., Haley, C. B., McKusick, M. K., and Kessler, P. B., "Berkeley Pascal User's Manual", Version 3.0, Computer Science Division, EECS, University of California, Berkeley, CA 94720. July 1983.

[Karr84]
Karr, M., "Code Generation by Coagulation", Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices 19, 6, Montreal, Quebec. pp. 1-12, June 1984.

[Katevenis83]
Katevenis, M., Sherburne, R., and Sequin, C., "The RISC II Micro-Architecture", International Conference on Very Large Scale Integration (VLSI '83)", Trondheim, Norway. pp. 349-359. August 1983.

[Kernighan78]
Kernighan, B. W., and Ritchie, D. M., "The C Programming Language", Prentice Hall, Englewood Cliffs, NJ 1978.

[Kessler83]
Kessler, P. B., "The Intermediate Representation of the Portable C Compiler, as Used by the Berkeley Pascal Compiler", Unpublished Paper, Computer Science Division, EECS, University of California, Berkeley, CA 94720. April 1983.

[Knuth71]
Knuth, D. E., "An Empirical Study of FORTRAN Programs", Software - Practice and Experience, 1, 1. pp. 105-133. January 1971.

[Kornerup80]
Kornerup, P., Kristensen, B., and Madsen, O., "Interpretation and Code Generation Based on Intermediate Languages", Software - Practice and Experience, 10, 4. pp. 635-658. 1980.

[Lamb83]
Lamb, D., "Sharing Intermediate Representations: The Interface Description Language", PhD Dissertation, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., May 1983.

[Leavens84]
Leavens, G. T., "Bibliography on Data Types", ACM SIGPLAN Notices 19, 8. pp. 41-50. August 1984.

[Leverett79]
Leverett, B., Cattell, R., Hobbs, S., Newcomer, J., Reiner, A., Schatz, B., and Wulf, W., "An Overview of the Production Quality Compiler Compiler Project", Technical Report TR-79-105, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., 1979.

[Leverett81]
Leverett, B. W., "Machine Independent Register Allocation in Optimizing Compilers", PhD Dissertation, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., February 1981.

[Leverett82]
Leverett, B. W., "Topics in Code Generation and Register Allocation", Technical Report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., July 1982.

[Lions79]
Lions, J., "The Second Pass of the Portable C Compiler", Technical Report, Bell Laboratories, Murray Hill, NJ. June 1979.

[Lunell83]
Lunell, H., "Code Generation Writing Systems", Software Systems Research Center, S-58183 Linkoping, Sweden. 1983.

[McKusick83]
McKusick, M. K., "Inline Expansion of Language Primitives in the Berkeley Pascal Compiler", Unpublished notes, March 1983.

[Morgan82a]
Morgan, T. M., and Rowe, L. A., "Analyzing Exotic Instructions for a Retargetable

Code Generator". Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices 17, 6, Boston, MA. pp. 197-204. June 1982.

[Morgan82b]
Morgan, T. M., "Analysis Techniques for Exotic Machine Instructions and Their Use in Retargetable Compilers", PhD Dissertation, Computer Science Division, EECS, University of California, Berkeley, CA 94720. June 1982.

[Motorola79]
Motorola Inc. "MC68000 16-Bit Microprocessor User's Manual", Prentice-Hall, Inc., Englewood Cliffs, NJ 07632. 1979, 1980, 1982.

[Ottenstein84]
Ottenstein, K. J., "Intermediate Program Representations in Compiler Construction: A Supplemental Bibliography", ACM SIGPLAN Notices 19, 7. pp. 25-27. July 1984.

[Patterson81]
Patterson, D. A., and Sequin, C. H., "RISC I: A Reduced Instruction Set VLSI Computer", Proceedings of the Eighth Symposium on Computer Architecture, Minneapolis, MN. pp. 443-457. May 1981.

[Pelegri-Llopart84]
Pelegri-Llopart, E., "Tree Transformation Systems in CODEGEN", Unpublished Qualifying Examination Proposal, Computer Science Division, EECS, University of California, Berkeley, CA 94720. Sept 1984.

[Radin82]
Radin, G., "The 801 Minicomputer", Proceedings of the ACM Symposium on Architectural Support for Programming Languages and Operating Systems, SIGPLAN Notices 17, 4. Palo Alto, CA. pp. 39-47. March 1982.

[Ripken77]
Ripken, K., "Formale Beschreibung von Maschinen, Implementierungen und optimierender Maschinencodeerzeugung aus attributierten Programmgraphen", PhD Dissertation, Technische Universitat Munchen, West Germany. July 1977. Summary Translation by [Speelpenning77]

[Schulman81]
Schulman, R., "A VAX Code Generator" Master's Project Report, Computer Science Department, University of California, Berkeley, CA. June 1981.

[Sethi70]
Sethi, R. and Ullman, J. D., "The Generation of Optimal Code for Expressions", JACM 17, 4. pp. 715-728. 1970.

[Speelpenning77]
Speelpenning, B., "A Review of Ripken's Thesis" Unpublished Manuscript, December 1977.

[Steel61]
Steel, T. B., "A First Version of UNCOL", Proceedings of the Western Joint Computer Conference 19, pp. 371-378. 1961.

[Steele80]
Steele, G., and Sussman, G., "The Dream of a Lifetime: A Lazy Variable Extent Mechanism", Conference Record of the 1980 LISP Conference, Stanford University, Stanford, CA. pp. 163-172. August 25-27, 1980.

[Strong58]
Strong, J. "The Problem of Programming Communication with Changing Machines: A Proposed Solution", CACM 1, 8. pp. 12-18. August 1958.

[Weicker84]
Weicker, R. P., "Dhrystone: A Synthetic Systems Programming Benchmark", CACM 27, 10. pp. 1013-1030. October 1984.

[Wirth71]
Wirth, N., "The Programming Language Pascal", Acta Informatica, Springer-Verlag, Vol. 1, No. 1, pp. 35-63. 1971.

[Wulf75]
Wulf, W., Johnsson, R., Weinstock, C., Hobbs, S., and Geschke, C., "The Design of an Optimizing Compiler", Elsevier North-Holland, Inc., New York, 1975.

[Wulf80]
Wulf, W., "PQCC: A Machine Relative Compiler Technology", Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., September 1980.