Code Generation Techniques

Todd Alan Proebsting

Technical Report #1119

November 1992

CODE GENERATION TECHNIQUES

By

Todd Alan Proebsting

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1992

Abstract

Optimal instruction scheduling and register allocation are NP-complete problems that require heuristic solutions. By restricting the problem of register allocation and instruction scheduling for delayed-load architectures to expression trees we are able to find optimal schedules quickly. This thesis presents a fast, optimal code scheduling algorithm for processors with a delayed load of 1 instruction cycle. The algorithm minimizes both execution time and register use and runs in time proportional to the size of the expression tree. In addition, the algorithm is simple; it fits on one page.

The dominant paradigm in modern global register allocation is graph coloring. Unlike graph-coloring, our technique, *Probabilistic Register Allocation*, is unique in its ability to quantify the likelihood that a particular value might actually be allocated a register *before* allocation actually completes. By computing the likelihood that a value will be assigned a register by a register allocator, register candidates that are competing heavily for scarce registers can be isolated from those that have less competition. Probabilities allow the register allocator to concentrate its efforts where benefit is high *and* the likelihood of a successful allocation is also high. Probabilistic register allocation also avoids backtracking and complicated live-range splitting heuristics that plague graph-coloring algorithms.

Optimal algorithms for instruction selection in tree-structured intermediate representations rely on dynamic programming techniques. Bottom-Up Rewrite System (BURS) technology produces extremely fast code generators by doing all possible dynamic programming before code generation. Thus, the dynamic programming process can be very slow. To make BURS technology more attractive, much effort has gone into reducing the time to produce BURS code generators. Current techniques often require a significant amount of time to process a complex machine description (over 10 minutes on a fast workstation). This thesis presents an improved, faster BURS table generation algorithm that makes BURS technology more attractive for instruction selection. The optimized techniques have increased the speed to generate BURS code generators by a factor of 10 to 30. In addition, the algorithms simplify previous techniques, and were implemented in fewer than 2000 lines of C.

Acknowledgements

I have benefited from the help and support of many people while attending the University of Wisconsin. They deserve my thanks.

My mother encouraged me to pursue a PhD, and supported me, in too many ways to list, throughout the process.

Professor Charles Fischer, my advisor, generously shared his time, guidance, and ideas with me.

Professors Susan Horwitz and James Larus patiently read (and re-read) my thesis.

Chris Fraser's zealous quest for small, simple and fast programs was a welcome change from the prevailing trend towards bloated, complex and slow software.

Robert Henry explained his early BURS research and made his Codegen system available to me.

Lorenz Huelsbergen distracted me with enough creative research ideas to keep graduate school fun.

National Science Foundation grant CCR–8908355 provided my financial support. Some computer resources were obtained through Digital Equipment Corporation External Research Grant 48428.

Contents

Chapter 1

Introduction

> *The three main problems in code generation are what instructions to use, in what order to do the computations, and what values to keep in registers.*
>
> Aho, Johnson, and Ullman [AJU77].

1.1 Overview

This thesis describes the following issues in code generation theory and technology: Chapter 2 develops an optimal instruction scheduler and register allocator for delayed-load architectures, Chapter 3 describes *probabilistic register allocation*, a new global register allocation heuristic that is simpler and more effective than widely-used graph-coloring techniques, and Chapter 4 outlines the design and implementation of new and highly optimized techniques for producing retargetable BURS instruction selectors. Appendix A is a reference manual for BURG, the code generator-generator system developed using the techniques described in Chapter 4.

1.2 Instruction Scheduling

The *instruction scheduling* phase of code generation determines the order in which program instructions will execute. For modern machine architectures with pipeline constraints this phase of compilation is essential to generating efficient code. A common pipeline constraint on reduced instruction set computer (RISC) architectures is that a value loaded from memory into a register will not be available for use for some number of subsequent cycles. During the intervening cycles it is important to schedule other instructions to execute that do not rely on that loaded value.

Machines with such load constraints are called *delayed-load architectures.* RISC chips like the MIPS R2000 and the SPARC are both delayed-load architectures that require a single additional cycle before a loaded value is available in its target register. For these two machines, it is necessary to find one instruction that can execute immediately after the load instruction (and does not rely on the loaded value). If no such instruction can be found, then it is necessary to *do nothing* for that cycle—thereby wasting it. On the R2000, it is necessary to put an explicit **NOP** in the instruction stream; on the SPARC, the pipeline will automatically interlock and stall the processor for the additional cycle.

Previously, most instruction schedulers handled delayed-load scheduling by solving a more general problem of arbitrary instruction scheduling ([HG83], [GM86], [War90], [LLM+87], and [PS90]). Arbitrary instruction scheduling considers operations other than loads with delays (such as multiplies/divides) that can take many cycles to complete. This thesis describes the *delayed-load scheduling* (DLS) algorithm for doing instruction scheduling for computations in which the only instructions facing pipeline constraints are loads. The more general algorithms are heuristic and typically take $O(n^2)$ time (where n is the number of instructions to be scheduled). DLS, on the other

hand, takes only $O(n)$ time, and is optimal when the load delay is one cycle. When the delay is greater than one cycle, DLS works as an excellent heuristic.

Most of the more general scheduling algorithms do not attempt to do register allocation along with scheduling. They either schedule before allocation or afterwards, but the two issues are considered separately. This phase-ordering problem can cause the phase that comes second to work poorly. If scheduling comes first, registers tend to be over-allocated. If allocation comes first, the number of possible schedules (given that allocation) can be quite small, and may not include any good schedules. DLS, on the other hand, integrates instruction scheduling with register allocation. Done together, DLS is able to find the optimal schedule that uses the *minimum* number of registers for any optimal schedule.

Chapter 2 describes the delayed-load scheduling algorithm, DLS. In addition to the algorithm, the chapter includes an optimality proof for machines with delay of one, empirical results when the delay is greater than one, and extensions to handle expressions that require more than the available number of registers.

1.3 Register Allocation

Instructions that access operands in registers are usually cheaper than those that access operands in memory. Many instructions cannot access memory directly: if an operand is in memory, it is necessary to explicitly load that operand for use. In either case, it is advantageous to keep frequently used operands in registers.

Registers, however, are scarce. Even RISC processors, which are often characterized as having many registers, typically have fewer than 32 general purpose registers. Often, in a computer program, there are many more values that are candidates to go in registers

than there are registers. The *register allocation* phase of code generation is responsible for determining which values can most advantageously be held in registers and at what points in the program.

Estimating the benefit of allocating a register to a value is not difficult. Execution frequency estimates for each instruction accessing the value are simply multiplied by the potential savings at each instruction. What is more difficult to estimate is how an allocation will affect subsequent allocation decisions.

If a value is allocated a register over a given set of program points, no other value that is simultaneously live can be allocated the same register. Each allocation made by a register allocator is made at the expense of other candidates. While there may be a candidate that will yield the greatest single immediate benefit, it may be the case that it will take a register from many other candidates whose aggregate benefit would have been greater.

Previous global register allocation methods have concentrated on casting register allocation as a graph-coloring problem ([CAC+81], [CH90], [BCKT89], [LH86]). Since no two simultaneously live values may be assigned to the same register, an *interference graph* can be built where nodes represent register candidate values, and arcs exist between simultaneously live values. Heuristics are used to find an assignment of the available registers to the nodes such that two connected nodes are not assigned the same register. Unfortunately, while graph coloring ensures a legal assignment, it does not accurately measure how the different (connected) nodes are competing—only that they are competing—for registers.

To more accurately measure the effects of allocation of a register to one value over another, a *probabilistic* measure can be made. Our technique, *probabilistic register*

allocation, quantifies the likelihood that a particular value might actually be allocated a register *before* allocation completes. By computing the likelihood that a value will be assigned a register by a register allocator, register candidates that are competing heavily for scarce registers can be isolated from those that face lower competition. These probabilities allow the register allocator to concentrate its efforts where benefit is high and where the competition for registers is low.

The implementation of a register allocator that exploits these probabilistic estimates has proven to be very successful in finding profitable candidates for register allocation while weighing both the benefit of the allocation, and the effects of such an allocation.

1.4 Instruction Selection

Instruction scheduling and register allocation often assume that instructions have already been chosen to do the computations required. The *instruction selection* phase of compilation determines which instructions can best do those computations. Typically, instructions are selected to minimize the size of the generated code, or to minimize the execution time. The instruction sets of most modern CPU's are redundant. That is, there are computations that may be evaluated via two or more different sequences of instructions. The instruction selector must choose among the various (correct) options to produce the best code.

Machine architectures are not trivial, and it is not always obvious what code will most cheaply evaluate some expression. Many complex instruction set computer (CISC) architectures have many addressing modes, each of which may subsume some number of additions and shifts. Simply recognizing where they are applicable may seem difficult. Furthermore, comparing all possible combinations that legally compute the desired

result might seem computationally intensive.

Fortunately, if we restrict our attention to expression trees (rather than the more general directed acyclic graphs) selecting optimal instructions is a straightforward problem. To do this, we will express the instruction set of the machine as a set of tree patterns. If the tree patterns that describe different instructions are given weights to describe their relative costs, dynamic programming can be used to select the optimal set of instructions to evaluate the tree ([AGT89], [AJ76], [PLG88], [BDB90], and [AG85]).

Dynamic programming is an expensive operation since it finds all optimal subsolutions before finding a solution for the entire tree. Fortunately, Bottom-Up Rewrite System (BURS) technology, can hide this cost from the compiler [PLG88]. BURS technology *pre-processes* the tree patterns and their costs to build automata that can drive instruction selection very quickly. BURS generated instruction selectors can be built that execute fewer than 50 VAX instructions per node of an expression tree [FH91c].

BURS code generators are fast for two reasons: they use bottom-up tree pattern matching technology (the theoretically fastest possible [HO82]), and they do all dynamic programming at compile-compile time (*i.e.*, when the patterns are pre-processed to build the code generator). By doing dynamic programming at compile-compile time, a BURS code generator can anticipate all possible input trees with information stored in tables. An enormous amount of computation is necessary to do dynamic programming in anticipation of all possible trees. It is, therefore, important to have an efficient BURS automata generator.

Chapter 4 describes an extremely fast BURS automata generator. The algorithm described is a work-list algorithm that employs simple optimizations. The implementation of the simple algorithm is a code generator generator, BURG, that runs 10 to

30 times faster than the previous best system [FHP91]. That increase in speed lowered the time to pre-process a VAX grammar from over 7 minutes to under 15 *seconds* on a DECstation 5000.

Since its development, BURG has been made publicly available, and is being used at AT&T Bell Labs to develop code generators for an ANSI C compiler [FH91b].

Chapter 2

Delayed-Load Scheduling

Modern RISC architectures are characterized by small, simple instruction sets, and general-purpose registers. While simple functionally, many of the instructions are complicated by instruction scheduling requirements. For instance, on a MIPS R2000, an integer load from memory into a register requires a single delay cycle before the loaded value can be accessed. It is necessary to find another instruction—that does not rely on the loaded value, or contribute to the load's address computation—to be placed immediately after the load. If no useful instruction can be found, it is necessary to put a NOP after the load to absorb the *delay* cycle.

Figure 2.1 gives two legal MIPS R2000 code sequences for evaluating `(a+b)+c`. The instructions selected to evaluate the expression are the same except for register assignment. The useful instructions differ only in their schedules (orders) and numbers of registers used. The right sequence requires two NOP's because the values loaded are accessed by the subsequent instructions. A compiler (or assembler) must order instructions carefully to minimize the costs of scheduling constraints.

While the optimal evaluation order in Figure 2.1 requires two fewer instructions than

Optimal	Non-Optimal
`ld r1, a`	`ld r1, a`
`ld r2, b`	`ld r2, b`
`ld r3, c`	`nop`
`addi r1, r1, r2`	`addi r1, r1, r2`
`addi r1, r1, r3`	`ld r2, c`
-	`nop`
-	`addi r1, r1, r2`

Figure 2.1: Two Legal Schedules to Evaluate `(a+b)+c` on a MIPS R2000.

the non-optimal, it does require one more register. Avoiding scheduling conflicts requires the ability to move operations *away* from the instructions that load their operands. This lengthens the span of those register operands, and, therefore, increases the number of registers in use. Because registers are scarce, and can be advantageously used to hold temporary and global values, it is important not to overuse them when scheduling instructions.

2.1 Overview

The problem of optimally scheduling instructions under arbitrary pipeline constraints is NP-complete ([GJ79], [LLM+87], [HG82], and [PS90]). Many heuristics have been proposed for scheduling pipelined code; all assume, however, that pipeline constraints can occur after any instruction, and that operators may share common subexpressions. The intractability of finding an optimal schedule holds even if an unlimited number of registers is available. Optimal local register allocation in itself is also NP-complete in the presence of common subexpressions [GJ79]. Such negative results have led to the belief that generating good quality code for RISC machines with pipeline constraints is too difficult to do well except in complex optimizing compilers.

Fast, optimal algorithms, however, can be devised for simpler, yet realistic architectures. Our results show that for a restricted set of pipeline constraints and a simple RISC load/store architecture, optimal code can be generated in linear time for expressions without operand sharing. Our *delayed-load scheduling* algorithm, *DLS*, efficiently combines instruction scheduling and register allocation. It is restricted to handling expression trees in which all leaf nodes are direct memory references. DLS is as an attractive, simple, fast and effective alternative to more complicated, slower heuristic solutions.

2.2 Previous Work

An adaptation of Hu's algorithm [Hu61] gives an optimal solution to scheduling a tree-structured task system on multiple identical processors if each task has unit execution time [Cof76], but the algorithm does not handle register allocation constraints. For an architecture with 2 functional units, one for loads and one for operations, with identical pipeline constraints, Bernstein *et. al.* have investigated code scheduling with register allocation for trees ([BPR84] and [BJR89]). Although applicable to a much different machine, Bernstein's results and algorithms are similar to ours[1]—both minimize pipeline interlocks and register usage, and both run in $O(n)$ time (where n is the number of nodes in the expression).

Code scheduling algorithms and heuristics for pipelined architectures have been extensively studied in recent years. Most of the attention to code scheduling has been

[1]Ours can issue only one instruction per cycle.

directed at scheduling expressions represented by directed acyclic graphs (DAGs) for architectures with pipeline constraints after both loads and operations.[2] Heuristic attacks on this general problem can be found in [HG82], [HG83], [GM86], [War90], [LLM+87], and [PS90]. These techniques are similar in spirit; they schedule instructions from the bottom of the DAG based on differing priority heuristics. The heuristics tend to favor those instructions that (a) are ready to execute (*i.e.*, do not face pipeline constraints); (b) will cause subsequent pipeline constraints (*i.e.*, need to be scheduled early); (c) are "far" from the roots of the DAG (*i.e.*, may be on a critical execution path).

Many heuristic solutions treat register allocation as a separate issue that occurs either before or after scheduling. Most heuristics work in a breadth-first manner from the bottom of the DAG up, they tend to cause many values to be live at once—filling up scarce registers. Unlike DLS, these algorithms fail to fully integrate code scheduling and register allocation, and therefore suffer from phase-ordering problems. In addition, whereas DLS runs in $O(n)$ time, these algorithms run in $O(n^2)$ time, and must have an additional register allocation phase.

Attempts to integrate register allocation and scheduling have been made at the basic block level. The techniques express the data dependences between instructions within a basic block as a DAG. Given the DAG, they attempt to schedule the instructions while both obeying pipeline constraints and minimizing registers. Since both optimal scheduling and register allocation on DAGs are NP-complete problems, their solutions to the integrated problem are heuristic.

Goodman and Hsu [GH88] describe a system, Integrated Prepass Scheduling (IPS), that combines register allocation and instruction scheduling. IPS is conceptually simple.

[2] We will use *operations* to denote non-load instructions.

The input is an instruction DAG for which registers have not been assigned. IPS consists of two possible schedulers: CSP, and CSR. CSP does heuristic scheduling at the cost of voracious register use, and CSR tends to minimize register use while possibly doing poor scheduling. Given a DAG, IPS schedules instructions using CSP and maintains a count of live registers. When the count exceeds a threshold, IPS switches to CSR to reduce register usage. Once reduced appropriately, IPS reverts to CSP. This oscillation continues until the scheduling process is complete.

Bradlee, Eggers, and Henry [BEH91] describe another integrated system, Register Allocation with Schedule Estimates (RASE), and compare it to IPS. RASE works in three sequential passes: PRESCHED, GRA, and FINALSCHED. For each basic block, PRESCHED estimates the cost of evaluating that basic block with n registers available, for all legal register counts. Given these cost vectors, the global register allocator, GRA, computes the optimal number of registers to give to the block in face of register competition for global values. FINALSCHED simply completes the schedule required by the register level determined by GRA.

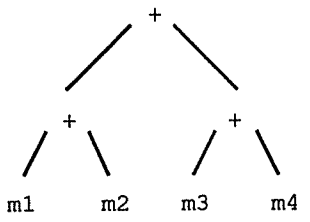
Bradlee found that IPS and RASE work well in practice—reducing execution time by an average of 12%. While RASE occasionally worked better, the resulting improvement was not significant. Both systems rely on heuristic scheduling techniques that are slow ($O(n^2)$), and require an *ad hoc* integration of register allocation and instruction scheduling.

2.3 Delayed-Load Architecture

We restrict our attention to a simple class of architectures—RISC load/store architectures with delayed loads. All instructions require a single instruction cycle to issue, and

reg	←	*memory*	**load** *memory*, *reg*
reg_i	←	reg_j **op** reg_k	**op** reg_j, reg_k, reg_i
memory	←	*reg*	**store** *reg*, *memory*

Figure 2.2: DLS Machine Model



Cycle#	With Interlocks	Without Interlocks
1.	**load** m1, r1	**load** m1, r1
2.	**load** m2, r2	**load** m2, r2
3.		**load** m3, r3
4.	**add** r1, r2*, r2	**load** m4, r4
5.	**load** m3, r1	**add** r1, r2, r2
6.	**load** m4, r3	**add** r3, r4, r4
7.		**add** r2, r4, r4
8.	**add** r1, r3*, r3	
9.	**add** r2, r3, r3	

Figure 2.3: Sample Expression Tree and Two Evaluation Sequences

only loads are pipelined. Our simple machine's instruction set is given in Figure 2.2. This architecture is an approximation of the integer functional units of many modern RISC processors such as the SPARC and MIPS R3000 [PH90].

A delayed load requires that the destination of a load not be accessed by subsequent instructions for some number of instruction cycles, although other, unrelated instructions may execute. `Delay` will be used to refer to the number of cycles that must elapse before the destination register is ready to be used. An attempt to use a destination register prior to the elapsing of `Delay` cycles forces a pipeline interlock that blocks processor execution until the register has finished loading.

Figure 2.3 shows two possible evaluations of an example expression tree. It is assumed that `Delay=1`. The (naively produced) left sequence wastes cycles due to pipeline interlocks at times 3 and 7—asterisks (*) denote the registers with which the delays are associated. The right sequence incurs no delays.

2.4 Register Allocation Trade-offs

Register allocation and instruction scheduling interact because the order of instructions determines the register needs for computing a given expression. Likewise, register allocation can limit or expand the possibilities for re-ordering code to limit pipeline interlocks.

If register allocation precedes instruction scheduling, the ability to schedule the code can be severely limited by constraints induced not by data dependences, but by constraints introduced by potential register interference. If register allocation follows instruction scheduling, a given schedule may require unnecessarily many registers, thus limiting the effectiveness of a global optimizer and possibly requiring spill code. This well-known phase-ordering problem is accepted in practice, but can lead to sub-optimal register use because the instruction schedulers minimize interlocks *without* taking into account the possibility that increased register demands could lead to costly register spilling.

The DLS algorithm avoids this phase-ordering problem by scheduling code and allocating registers in tandem. DLS optimally schedules instructions to avoid all delayed-load interlocks for expression trees when `Delay=1`. Furthermore, it finds an interlock-free schedule that minimizes register usage. When `Delay>1` or when DAGs are transformed into trees, DLS serves as an excellent heuristic while retaining its conceptual simplicity, guaranteed linear performance, and integrated register allocation.

2.4.1 Canonical Form

Generating code and allocating registers is much simpler for expression trees than for arbitrary DAGs. Once a preliminary schedule for the code has been generated for a

tree, and the register needs determined, it is possible to reschedule the code and reassign the registers to obtain a code sequence in a canonical form. This canonical form has three important invariants: the relative order of the operators remains unchanged, the relative order of the loads remains unchanged, and the number of registers needed remains unchanged. For a given number of registers and specific operation and load orders, the canonical order will minimize pipeline interlocks for a delayed-load machine.

The canonical schedule is produced by moving loads as early as possible in the initial instruction sequence (subject to the three invariants). Shifting the loads will move a load away from its parent in the tree and therefore increase the number of instructions between the load and its dependent operation.

To produce the canonical ordering of an instruction sequence using R registers that has L loads and (L–1) operations,[3] create an ordering that consists of R loads followed by an alternating sequence of L–R (op,load) pairs, followed by the remaining R–1 operations. Loads are moved before operations that they had previously followed—this does not affect data dependences since all operations depend on registers and all loads depend on memory. The movement of the loads relative to the operations will cause the necessary register assignments to change; if done systematically this will not cause the register needs to increase. Since loads increase the number of registers in use by one, and operations decrease the number of registers in use by one, the number of registers in use at any point in the evaluation is equal to the number of loads performed minus the number of operations performed. A canonical order evaluation, therefore, ensures that the number of registers in use will never exceed R.

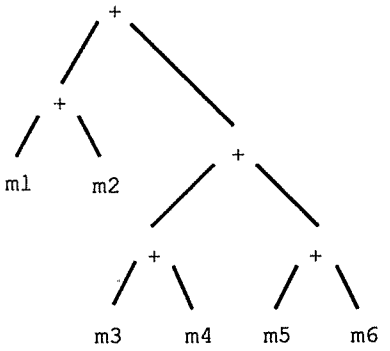
[3]There are (L–1) operations in a binary tree with L loads.

Figure 2.4 gives an example expression with a standard Sethi-Ullman (SU) instruction schedule [SU70], a canonical order with 3 registers, and a canonical order assuming 4 registers. (The Sethi-Ullman order, which is optimal with respect to register usage, orders instructions by scheduling sub-trees separately so that the sub-tree with the greater register needs is scheduled first.) Simply putting the SU-generated instructions into canonical form without additional registers removes 1 pipeline interlock. Adding the extra register eliminates all interlocks. Note that the relative orders of loads and the relative order of operations is the same in all three sequences.

2.4.2 Adding Registers Helps

As is seen in Figure 2.4, adding registers sometimes helps. This follows from the observation that loads can often be *shifted* backwards (*i.e.*, earlier) in the instruction sequence without affecting the outcome of the computation. This shifting does not change the relative ordering of the loads with respect to one another, or the relative ordering of the operations with respect to one another—it simply shifts the loads farther from the operations that use them. Shifting a load farther away allows its delay slot to be filled with an intervening load or operation.

Minimizing the number of registers needed to evaluate an expression without load delays is an essential consideration. If the operations in the expression tree in Figure 2.4 were evaluated from left to right, it would be necessary to use 5 registers rather than 4 to produce a canonical evaluation without interlocks. It is therefore necessary to treat the problem of optimal code generation as one of minimizing pipeline interlocks and register usage through code scheduling.



#	Sethi-Ullman(3)	Canonical(3)	Canonical(4)
1.	**load** m3, r1	**load** m3, r1	**load** m3, r1
2.	**load** m4, r2	**load** m4, r2	**load** m4, r2
3.		**load** m5, r3	**load** m5, r3
4.	**add** r1, r2*, r2	**add** r1, r2, r2	**load** m6, r4
5.	**load** m5, r1	**load** m6, r1	**add** r1, r2, r2
6.	**load** m6, r3		**load** m1, r1
7.		**add** r3, r1*, r1	**add** r3, r4, r4
8.	**add** r1, r3*, r3	**load** m1, r3	**load** m2, r3
9.	**add** r2, r3, r3	**add** r2, r1, r1	**add** r2, r4, r4
10.	**load** m1, r1	**load** m2, r2	**add** r1, r3, r3
11.	**load** m2, r2		**add** r4, r3, r3
12.		**add** r3, r2*, r2	
13.	**add** r1, r2*, r2	**add** r1, r2, r2	
14.	**add** r3, r2, r2		

Figure 2.4: Expression Tree and Canonical Instruction Sequences

2.5 Optimal Algorithm for `Delay=1`

Optimal instruction scheduling and register allocation for an expression tree when `Delay=1` can be done in time proportional to the size of the expression tree. Our DLS algorithm is a variation of the Sethi-Ullman algorithm adapted to our machine model. Both the SU algorithm and the DLS algorithm are driven by minimizing the register needs for evaluating an expression. These needs are denoted as the `minReg` of a node and refer to the minimal number of registers needed for computing the sub-tree

rooted at that node without spilling. The minReg value of a node is simply the standard SU number, adapted to our load/store architecture.[4] It is calculated by the following procedure, label().

```
procedure label(node :  ExprNode) {
     if (isLeaf(node)) {
          node.minReg = 1;
     } else {
          label(node.left);
          label(node.right);
          if (node.left.minReg == node.right.minReg) {
               node.minReg = node.left.minReg+1;
          } else {
               node.minReg = MAX(node.left.minReg, node.right.minReg);
          }
     }
}
```

The order of operations for an expression tree determines the optimal order of the loads—the loads will appear in the same relative order as their parents. This follows because forcing two (load,op) pairs, (l_i, op_i) and (l_k, op_k) (assuming that op_i comes before op_k), out of order would force the separation between (l_i, op_i) to be less than it was originally and less than the original (l_k, op_k) distance. This decrease in separation could cause pipeline interlocks. The increase in (l_k, op_k) separation may avoid some interlocks, but the net effect cannot be advantageous. The goal of finding the optimal instruction schedule and register usage therefore reduces to finding the optimal *operation* schedule and register usage.

[4]The original SU algorithm was based on a machine model in which binary operations could access their right operands directly from memory. Our model requires all operands to be in registers.

2.5.1 Exceptional Cases for `Delay=1`

When `Delay=1`, exactly two trees in our model have schedules that must always incur pipeline interlocks: the tree consisting of a single node, and the tree consisting of a single operator and two leaf (memory) nodes. It is trivial to verify that these must incur pipeline interlocks, and that the register needs for these trees are 1 and 2, respectively.

2.5.2 Algorithm

The DLS algorithm presented in Figure 2.5 finds an instruction schedule and register assignment that is optimal for a given expression tree. For all trees with the exception of the two just mentioned, the DLS schedule will have no pipeline interlocks and will use the minimal number of registers for *any* schedule without interlocks.

The number of registers needed for such a schedule is exactly one more than the minimal number of registers needed to evaluate the expression without any spills (*i.e.*, the SU `minReg` value of the root of the expression).

The DLS algorithm is a simple three-pass algorithm for finding the optimal instruction sequence and register allocation. The procedure `label()` (given earlier) labels the nodes with their SU `minReg` values. Procedure `order()` finds the operation and load orders. `order()` is similar to the original Sethi-Ullman algorithm. `schedule()` then emits the instructions in canonical order.

2.5.3 Optimality Proof

The argument that the DLS algorithm creates an optimal instruction schedule and register allocation follows from two observations: the number of registers to avoid interlocks must be at least `minReg+1`, and the canonical order generated by the algorithm using

```
// Sethi-Ullman Ordering
procedure order(root : ExprNode; var opSched, loadSched : NodeList) {
    if (not isLeaf(root)) {
        if (root.left.minReg < root.right.minReg) {
            order(root.right, opSched, loadSched);
            order(root.left, opSched, loadSched);
        } else {
            order(root.left, opSched, loadSched);
            order(root.right, opSched, loadSched);
        }
        append(root, opSched);
    } else {
        append(root, loadSched);
    }
}

// Canonical Ordering
procedure schedule(opSched, loadSched : NodeList; Regs : integer) {
    initialLoads : integer = MIN(Regs, length(loadSched));
    for i = 1 to initialLoads do {          // Loads First
        ld = popHead(loadSched);
        ld.reg = getReg();
        gen(Load, ld.name, ld.reg);
    }
    while (not Empty(loadSched)) {          // (Operation,Load) Pairs
        op = popHead(opSched);
        op.reg = op.right.reg;
        gen(op.op, op.left.reg, op.right.reg, op.reg);
        ld = popHead(loadSched);
        ld.reg = op.left.reg;
        gen(Load, ld.name, ld.reg);
    }
    while (not Empty(opSched)) {           // Remaining Operations
        op = popHead(opSched);
        op.reg = op.right.reg;
        gen(op.op, op.left.reg, op.right.reg, op.reg);
        freeReg(op.left.reg);
    }
}

// Schedule Instructions
procedure generate(root : ExprNode; Delay : integer) {
    label(root)                                          // Compute minReg
    opSched = loadSched = emptyList();                   // Initialize
    order(root, opSched, loadSched);                     // Find Load/Op Order
    schedule(opSched, loadSched, root.minReg+Delay); // Emit Canonical Order
}
```

Figure 2.5: Optimal Delay Load Scheduling (DLS) Algorithm

`minReg+1` registers does not incur pipeline interlocks.

To incur no load delays requires at least `minReg+1` registers. The evaluation cannot take fewer than `minReg` registers by definition. If only `minReg` registers were available, there must be a point at which a just loaded register must be used in the next instruction, which would result in a load delay. This follows from the fact that only loads can increase the number of registers in use, and thus at some point a load must put `minReg` registers in use. Because only `minReg` registers are available, this load must be followed by an operation on the just loaded register (if another operation could have been scheduled, it would have been to keep the number of registers in use at a minimum). Therefore, more than `minReg` registers are needed to avoid interlocks.

Because `Delay=1`, it is only necessary to find a single instruction to fill every load delay slot. These can be other loads or (unrelated) operations. Adding another register, but keeping a Sethi-Ullman ordering for the operations and a canonical ordering for the entire instruction stream assures that there is at least one more register *live* than there is in an SU order. This register must have been made live by a load since the SU algorithm would have at most `minReg` registers live at any point. In other words, the loads are at least one step ahead of the operations that can use them. Since the delay slot is exactly one cycle long, and loads are at least one cycle ahead, the loads must always be followed by an unrelated instruction. This means no pipeline interlocks occur for any expression tree with at least 3 operands (except for the two previously mentioned exceptions).

The DLS algorithm creates an operation order that can be evaluated using exactly `minReg` registers because it creates a Sethi-Ullman ordering for the operations and loads. Because it schedules the instructions in canonical order using a single extra register, it must satisfy the necessary conditions specified in the previous paragraph. Therefore,

given a single extra register, the algorithm creates an instruction sequence optimal in evaluation time and register usage.

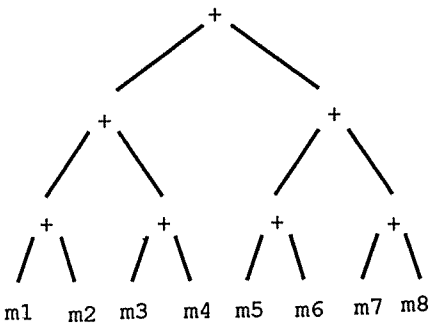
2.6 Spilling

For the DLS algorithm to be practical, it must also be able to produce good schedules when fewer than `minReg+1` registers are available for allocation. Suppose, for example, that exactly `minReg` registers are available. Should the algorithm introduce spill code so that sub-trees may be computed without interlocks? If so, where should the spills be introduced? If not, will the computation incur excessively many pipeline interlocks? The best solution depends on the form of the expression tree.

The tree in Figure 2.6 can be best handled by allowing the canonical execution order (with `minReg`=4 registers) and incurring a single load delay of 1 cycle. Having no interlocks would have required a spill and hence cost 2 extra instructions, a store and load. The tree in Figure 2.7 will incur 3 delay cycles, but would incur only the cost of a single load and store if a spill were introduced.

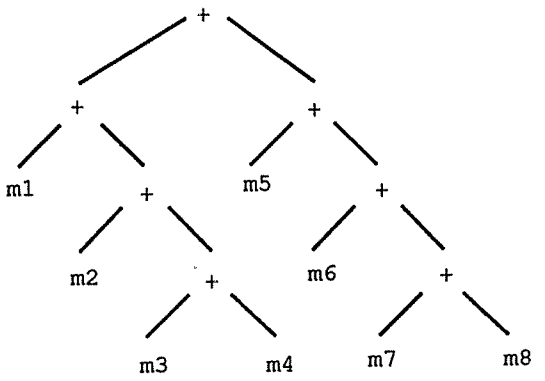
Ordering code so that spill/delay costs will be minimized requires extending the original DLS algorithm. Given an expression with two sub-trees that have identical `minReg` values, the algorithm orders the sub-trees such that the sub-tree exerting the minimal *register pressure* is scheduled last. The register pressure of a sub-tree is relative to the `minReg` of that sub-tree; it is simply a count of the times that `minReg` registers will be live in a normal SU evaluation order of the sub-tree. Register pressure is calculated by the routine in Figure 2.8.

The sub-trees of the tree in Figure 2.9 are labeled with their register pressures. The register pressure of the left sub-tree is 3 because its `minReg` value is 2, and in its



#	No Spill—Interlock	Spill—No Interlock
1.	**load** m1, r1	**load** m1, r1
2.	**load** m2, r2	**load** m2, r2
3.	**load** m3, r3	**load** m3, r3
4.	**load** m4, r4	**load** m4, r4
5.	**add** r1, r2, r2	**add** r1, r2, r2
6.	**load** m5, r1	**load** m5, r1
7.	**add** r3, r4, r4	**add** r3, r4, r4
8.	**load** m6, r3	**load** m6, r3
9.	**add** r2, r4, r4	**add** r2, r4, r4
10.	**load** m7, r2	**load** m7, r2
11.	**add** r1, r3, r3	**store** r4, TEMP
12.	**load** m8, r1	**load** m8, r4
13.		**add** r1, r3, r3
14.	**add** r2, r1*, r1	**load** TEMP, r1
15.	**add** r3, r1, r1	**add** r2, r4, r4
16.	**add** r4, r1, r1	**add** r3, r4, r4
17.		**add** r1, r4, r4

Figure 2.6: Spilling May be More Expensive than Interlocks—Example with 4 Registers



#	No Spill—Interlock	Spill—No Interlock
1.	load m3, r1	load m3, r1
2.	load m4, r2	load m4, r2
3.	load m2, r3	load m2, r3
4.	add r1, r2, r2	add r1, r2, r2
5.	load m1, r1	load m1, r1
6.	add r3, r2, r2	add r3, r2, r2
7.	load m7, r3	load m7, r3
8.	add r1, r2, r2	add r1, r2, r2
9.	load m8, r1	load m8, r1
10.		store r2, TEMP
11.	add r3, r1*, r1	load m6, r2
12.	load m6, r3	add r3, r1, r1
13.		load m5, r3
14.	add r1, r3*, r3	add r2, r1, r1
15.	load m5, r1	load TEMP, r2
16.		add r3, r1, r1
17.	add r3, r1*, r1	add r2, r1, r1
18.	add r2, r1, r1	

Figure 2.7: Spilling May Save Cycles—Example with 3 Registers

```
if (isLeaf(node)) {
    node.pressure = 1;
} else {
    if (node.left.minReg == node.right.minReg) {
        // Assumes that node with minimal pressure is scheduled last.
        node.pressure = MIN(node.left.pressure, node.right.pressure);
    } else if (node.left.minReg > node.right.minReg) {
        // node.left will be scheduled first -- Following standard SU rule.
        if (node.left.minReg == node.right.minReg+1) {
            // both subtrees contribute register pressure
            node.pressure = node.left.pressure + node.right.pressure;
        } else {
            // only the first subtree contributes register pressure
            node.pressure = node.left.pressure;
        }
    } else {
        if (node.right.minReg == node.left.minReg+1) {
            node.pressure = node.right.pressure + node.left.pressure;
        } else {
            node.pressure = node.right.pressure;
        }
    }
}
```

Figure 2.8: Computation of Register Pressure

evaluation 2 registers will be live 3 times—after loading `m3` (or `m4`), `m2`, and `m1`. The register pressure of the root node is only 1, however, because `minReg`=3 and the left sub-tree will be evaluated first (without ever having 3 registers live in a normal SU order), and the right sub-tree will be evaluated second, reaching 3 live registers only once—after `m5` (or `m6`). Therefore, the optimal evaluation of the tree (given 3 registers) will incur only one load delay by scheduling the left sub-tree before the right. Had the right been scheduled first, 3 load delays would have occurred.

The decision to spill a node is made when calculating the `minReg` and `pressure` values. A node is spilled if its `minReg` value is equal to the number of available registers

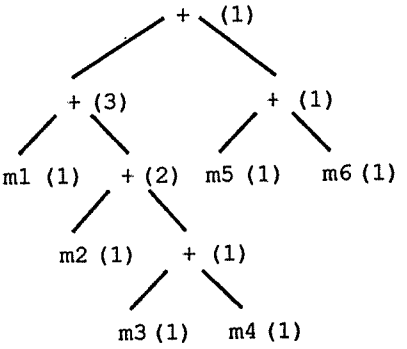


Figure 2.9: Tree with Register Pressure Labels

and its pressure is greater than 2 (the cost of a store and load). If a node has a minReg value greater than the number of available registers, then its child with the greater pressure should be spilled. Spilling information is calculated bottom-up in the tree while calculating minReg and pressure. The algorithm avoids spilling until absolutely necessary *or* until it is advantageous.

```
if ((node.minReg == Registers and node.pressure > 2)
   or node.minReg > Registers) {
      if (node.left.pressure > node.right.pressure) {
         // Spill node.left;
         // Make node.left a Leaf temporary;
         // Set node.left.pressure = 1;
         // Set node.left.minReg = 1;
         // Set node.pressure = node.right.pressure;
         // Set node.minReg = node.right.minReg;
      } else {
         // Spill node.right, etc...
      }
}
```

Loads introduced by spills will not cause pipeline interlocks because they will occur only at a node whose sibling has a minReg value of at least Registers-1. This ensures that the load will be part of a tree whose root has minReg of at least Registers-1. Since this new leaf (spill) node has a minReg of 1 and a register pressure of 1, it cannot increase

the minReg or register pressure of the entire tree. The cost of the spill is restricted to the cost of the store/load, and the interlocks associated with evaluating the sub-tree below the spilled node (which cannot be greater than 2).

2.7 Extensions to DLS: Related Work

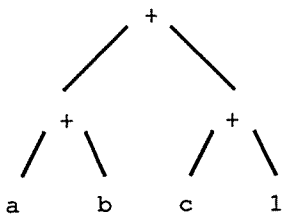
The DLS algorithm is restricted to considering binary trees with all leaves representing delayed loads. A more realistic machine model must be able to handle unary nodes, leaf instructions without delays, and delayed loads at internal nodes.

Kurlander, Fischer, and Proebsting [KFP92] have extended DLS to optimally handle unary nodes and non-delayed leaf nodes. The improvements are called Extended DLS (EDLS). In addition, they give a simple heuristic for scheduling trees with internal delayed loads.

2.7.1 Non-Delayed Leaf Nodes

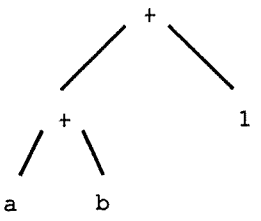
Not all leaf instructions on real machines incur delay cycles. For instance, the result of a "load immediate" is typically available immediately! Scheduling a tree containing a non-delayed leaf node using DLS will always give a non-interlocking schedule, but it may use too many registers. It is sometimes possible to schedule an expression with non-delayed leaf nodes using only minReg registers. Figure 2.10 shows that (a+b)+(c+1) can be scheduled (without interlocks) using exactly minReg= 3 registers. Figure 2.11 shows, however, that (a+b)+1 requires minReg+1 = 3 registers for a delay free schedule. The expression (a+b)+(c+1) could be scheduled without an extra register because the **loadi** of 1 could be *immediately* followed by an instruction using that value.

In the absence of non-delayed leaf nodes, the proof that an extra register is needed



#	Schedule
1.	**load** a, r1
2.	**load** b, r2
3.	**load** c, r3
4.	**add** r1, r2, r2
5.	**loadi** 1, r1
6.	**add** r3, r1, r1
7.	**add** r2, r1, r1

Figure 2.10: Delay-Free Schedule using only `minReg` registers.



#	Schedule
1.	**load** a, r1
2.	**load** b, r2
3.	**loadi** 1, r3
4.	**add** r1, r2, r2
5.	**add** r2, r3, r1

Figure 2.11: Delay-Free Schedule requiring `minReg+1` registers.

for delay-free evaluation followed because any schedule requiring only `minReg` registers would necessarily have load instructions that were immediately followed by operations using those loaded values. However, if *all* of the leaf instructions that increase the number of registers in use to `minReg` are non-delayed instructions, then the additional register is not necessary. This observation leads directly to a linear-time algorithm for determining if an additional register is needed. Every sub-tree will have a flag indicating whether or not that sub-tree can be evaluated optimally without an additional register beyond `minReg`. This can be computed in a single bottom-up pass of the tree (see Figure 2.12).

Recall that any (sub-)tree can be computed interlock-free with `minReg+1` registers (when `Delay=1`). If the left and right sub-trees each require an identical number of registers (`minReg`) to be computed, then the entire tree will be given `minReg+1` registers.

```
if (isLeaf(node)) {
   node.needExtra = isDelayed(node);
} else {
   if (node.right.minReg == node.left.minReg) {
      node.needExtra = node.right.needExtra AND node.left.needExtra;
   } else if (node.right.minReg < node.left.minReg) {
      if (node.right.minReg+1 == node.left.minReg) {
         node.needExtra = node.right.needExtra OR node.left.needExtra;
      } else {
         node.needExtra = node.left.needExtra;
      }
   } else { // node.right.minReg > node.left.minReg
      // Symmetric to right.minReg < left.minReg case.
   }
}
```

Figure 2.12: Computing the need for an Extra Register with Non-Delayed Leaf Nodes.

This means the sub-tree scheduled first will have `minReg+1` registers for its computation, and the DLS algorithm ensures that it will be computed without interlocks.[5] The second sub-tree will have only `minReg` registers. Therefore, if one sub-tree can be computed without an extra register, it can be computed after the other, and both will proceed interlock-free.

If the `minReg` values of two sub-trees differ, then their order is determined by normal Sethi-Ullman constraints—the sub-tree with the greater `minReg` value must be computed first. (The `minReg` value of the entire tree will be equal to that greater `minReg`.) Call the two sub-trees `left` and `right` and assume without loss of generality that `left.minReg` > `right.minReg`. `left` will be computed before `right`. If `left` needs an extra register,

[5]The two trees that do not have any delay-free schedules (a single load, and two loads and a single binary operation) *will not* incur delays when scheduled as the first of two sub-trees. The first instruction of the second sub-tree must fill the (otherwise unfilled) delay slot of the first sub-tree.

then the entire tree must need an extra register since the tree has the same `minReg` value as the left sub-tree. If `left` can be computed without an extra register, then it depends on whether or not `right` can be computed without interlocks given only `left.minReg-1` registers. If `right.minReg` is less than `left.minReg-1` then `right` will have the extra register it needs for DLS delay-free scheduling. If, however, `left.minReg-1` does not provide an extra register for `right` to be computed, then it is necessary to examine whether `right` needs an extra register (since it is **not** going to get one).

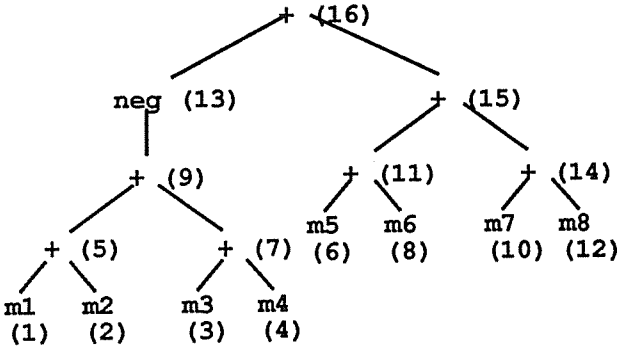
While this algorithm is simple and fast, its utility is debatable. At best it can save a *single* register. It does not find interlock-free schedules that DLS could not find given an extra register.

2.7.2 Unary Nodes

Unary operations can be scheduled via a simple extension to ordinary DLS. All unary nodes are scheduled immediately *before* the operations that are going to use the values they compute. Schedule an expression using DLS as if the unary nodes were collapsed into the operands of their parents, and then expand them after scheduling is complete.

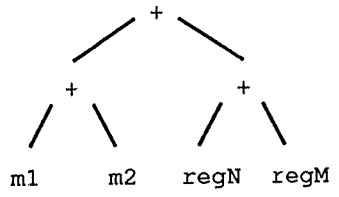
The (small) problem with this is that it too may overuse registers by one. It may be the case that a schedule exists for the expression that would not need `minReg+1` registers, but instead could use only `minReg`. Unfortunately, some of these schedules require a *non-contiguous* evaluation of the expression, and DLS cannot directly find such evaluations. Figure 2.13 gives an example of a tree that requires only `minReg`$= 4$ registers for an optimal schedule, but to do so it requires a non-contiguous evaluation.

Kurlander's linear-time algorithm to find the optimal schedules for trees with unary nodes is somewhat more complicated than the algorithm to handle non-delayed leaves.



#	Schedule
1.	**load** m1, r1
2.	**load** m2, r2
3.	**load** m3, r3
4.	**load** m4, r4
5.	**add** r1, r2, r2
6.	**load** m5, r1
7.	**add** r3, r4, r4
8.	**load** m6, r3
9.	**add** r2, r4, r4
10.	**load** m7, r2
11.	**add** r1, r3, r3
12.	**load** m8, r1
13.	**neg** r4, r4
14.	**add** r2, r1, r1
15.	**add** r3, r1, r1
16.	**add** r4, r1, r1

Figure 2.13: Non-Contiguous Evaluation with Unary Nodes



#	With Interlocks	Without Interlocks
1.	**load** m1, r1	**load** m1, r1
2.	**load** m2, r2	**load** m2, r2
3.		**add** regN, regM, r3
4.	**add** r1, r2*, r2	**add** r1, r2, r2
5.	**add** regN, regM, r3	**add** r2, r3, r2
6.	**add** r2, r3, r2	

Figure 2.14: Evaluation Orders with Register Variables

His algorithm is capable of finding the optimal non-contiguous evaluations. The detailed algorithm and proof of correctness can be found in [KFP92].

2.7.3 Register Variables

As presented, the DLS algorithm cannot handle register variables (*i.e.*, leaf nodes that do not represent load instructions). With register variables, it is not always the case that

leaf nodes allocate registers or that operations will decrease the number of registers in use. An operation that has two register variables for children will *increase* the number of registers in use when evaluated (assuming that the register variables are live after the expression).

The simple expression tree in Figure 2.14 demonstrates that a more powerful algorithm is needed to handle register variables. Normal Sethi-Ullman evaluation would label the left sub-tree with an SU-number of 2 and evaluate it before the right sub-tree with a SU-number of 1. The entire tree would therefore have a SU-number of 2. DLS would indicate that 3 registers would be needed for a delay-free evaluation, and that the left sub-tree's operator would be scheduled before the right's (a Sethi-Ullman ordering). Putting the operators and loads into a DLS-like canonical order gives an evaluation order in which an interlock will occur. Had the right sub-tree's operator been scheduled before the left's, no such interlock would have occurred.

Fortunately, Kurlander's results for unary and non-delayed leaf nodes subsumes the problem of handling register variables. For each operand of an operator that is a register variable, simply decrease the perceived arity of that operator by one. In Figure 2.14 the **add** of the two register variables should be treated as a non-delayed leaf node. It acts like a non-delayed leaf node since it has no associated delay, and it *increases* the number of temporary registers in use. Kurlander's scheduler would then find the optimal schedule.

2.7.4 DAGs, Forests, and Internal Loads

When code is generated or scheduled for an entire basic block it is natural to use a DAG as the intermediate representation because of common sub-expressions. Since optimal

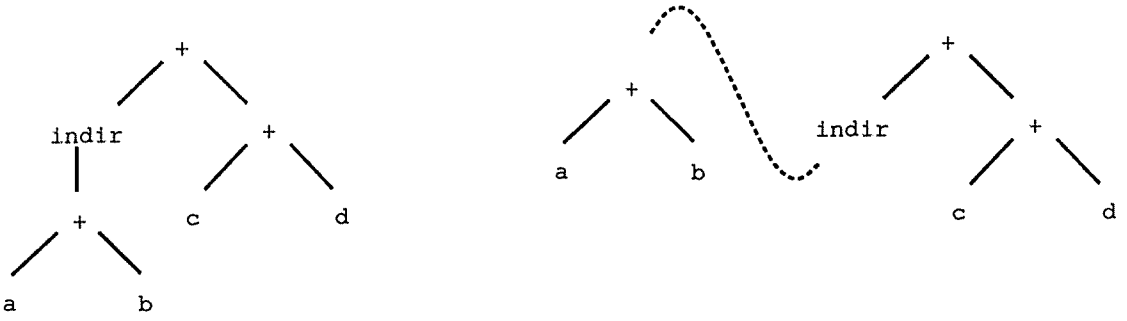


Figure 2.15: Splitting At Internal Loads

code scheduling for DAGs is NP-complete, we prefer to schedule trees. By *splitting* nodes of a DAG, it is possible to treat a DAG as an ordered forest of trees. A DAG is split by computing shared internal nodes to temporary storage prior to computing ancestor nodes. Proceeding in a bottom-up fashion, evaluating a DAG then reduces to evaluating a sequence of trees (a forest).

Furthermore, Kurlander [KFP92] has used this splitting idea to handle delayed loads that are internal to a tree (or DAG). Whenever a non-leaf load is encountered, the tree is split so that that load is now at the frontier of the original tree. Figure 2.15 shows the splitting as it would be applied to `(a+b)+indir(c+d)`. From the original tree, a *forest* of two trees has been created—each with loads only at the leaves. These trees can be handled by DLS and EDLS.

The idea of a canonical order for trees can be extended to forests with corresponding benefits. If two trees are executed in sequence and have no data dependences between them, it is possible to order their operators and loads separately, and then schedule both sets of operators and loads together. This is done by concatenating the operator lists together and the load lists together, and then scheduling these lists such that loads from the second expression are interspersed with the operators of the first expression.

#	Without Merging	With Merging	Comments
1.	**load** A, r1	**load** A, r1	
2.	**load** B, r2	**load** B, r2	
3.	**load** C, r3	**load** C, r3	
4.	**load** D, r4	**load** D, r4	
5.	**add** r1, r2, r2	**add** r1, r2, r2	
6.	**add** r3, r4, r4	**load** E, r1	Start merging A+B.
7.	**add** r2, r4, r4	**add** r3, r4, r4	
8.	**load** E, r1	**load** F, r2	
9.	**load** F, r2	**add** r2, r4, r4	
10.	{ NOP }	**add** r1, r2, r2	
11.	**add** r1, r2, r2	-	

Figure 2.16: Merging Two Trees in a Forest: (A+B)+(C+D); E+F

The operations of the first tree and the loads of the second tree are moved away from their respective loads and operations, possibly reducing register needs and pipeline interlocks. The right column of Figure 2.16 demonstrates that merging the expressions (A+B)+(C+D) and E+F yields a delay-free schedule requiring only 4 registers (the same number needed by (A+B)+(C+D) alone). This merged schedule avoids the interlock present when E+F is evaluated alone.

Entire basic blocks can also be handled in this fashion. Data dependences between stores and loads in a basic block will limit the ability to shift loads earlier in the schedule. The definition of a canonical order for a basic block must require that loads not be shifted before stores that may affect their value.

2.8 Behavior for Delay>1

The optimality results for `Delay=1` do not directly extend to greater `Delay` values. DLS is, however, an excellent heuristic for larger `Delay`'s, retaining its simplicity and linear running time. As a heuristic for instruction scheduling with `Delay>1`, DLS may require

more than `minReg+1` registers to achieve an interlock-free schedule in canonical order. If an interlock-free schedule exists for a given expression, it can be found, however, by using `minReg+Delay` registers with the DLS canonical form. The same argument made for the optimality of the case `Delay=1` shows that the number of registers needed for an interlock-free schedule will never be greater than `minReg+Delay`. Not all expressions require `minReg+Delay` registers for an interlock-free schedule—they may have interlock-free schedules requiring fewer registers. For this reason, a heuristic approximation to the DLS algorithm for `Delay>1` is to use a DLS-generated canonical order with `minReg+Delay` registers. This heuristic retains the optimal scheduling results of the `minReg+1` case, but may, in a few cases, over-allocate registers. In §2.8.2, we give the lower-bounds on the fewest possible registers needed for an interlock-free schedule when `Delay>1`.

2.8.1 Non-Contiguous Operand Ordering

The Sethi-Ullman algorithm generates code that is *contiguous*. That is, instructions generated for one sub-tree do not mix with the instructions for a sibling sub-tree. The DLS algorithm does not possess this property because the loads from one sub-tree may be mixed with the operations from another. The algorithm does, however, produce schedules that exhibit some contiguity: the loads taken alone, and the operations taken alone do have contiguous orders. It is precisely this property that allows a "divide and conquer" approach that treats each sub-tree separately.

It is not always possible to generate code with this contiguous operation/load property and still have the code be optimal with respect to pipeline interlocks and register usage if `Delay` is greater than 1. Figure 2.17 is the *smallest* example of a tree that does

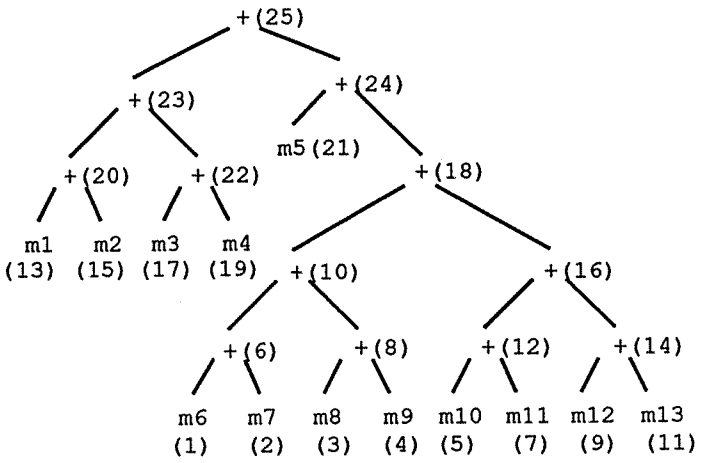


Figure 2.17: Non-Contiguous Optimal Evaluation for `Delay=2`.

not have an optimal schedule in which the operations/loads are ordered contiguously for `Delay=2`—the tree is labeled with the optimal evaluation order.

Lacking the contiguous property for optimal results, it is unlikely that a linear time algorithm exists for optimally scheduling trees with `Delay>1`. Whether the optimal algorithm is polynomial-time or exponential-time, it will be much more expensive to run than DLS (and in practice not all that more effective).

2.8.2 Register Bounds

The fact that `minReg+Delay` registers and a canonical ordering will always produce a delay-free schedule (when one exists) gives an upper-bound on the number of registers necessary to find such an evaluation. The optimality proof in §2.5.3 gives a lower (and upper) bound of `minReg+Delay` for the case where `Delay=1`. The previous section showed that `minReg+Delay` is *not* a lower bound when `Delay>` 1.

Given any Sethi-Ullman order evaluation of an expression, it is not difficult to show,

however, that minReg+⌈Delay/2⌉ is a lower bound on the registers needed for delay-free evaluation when put into canonical form (when Delay> 1). Given the canonical evaluation order with exactly minReg registers, it must be the case that some operation immediately follows the load of one of its operands. Adding a single register shifts the loads earlier, and operations later in the schedule. For each additional register, each load moves at most one instruction earlier in the schedule and each operation moves at most one instruction later. (Loads in the initial (non-alternating) sequence of loads do not move at all—likewise for the operations in the latter part of the schedule.) Thus, the addition of each register can move a load/operation pair at most two instructions further apart.

Therefore, if Delay instructions are necessary to fill a delay slot, and each register will move at most 2 instructions between any pair, it will be necessary to use a minimum of ⌈Delay/2⌉ registers in addition to the original minReg.

It is possible to find the minimal number of registers needed for a particular combination of operation and load orders in linear time. This can be done by computing the schedule (in canonical order) for each possible number of registers. Starting with minReg+Delay-1 registers and working down, the canonical orders are created and tested to make certain that the loads are separated from their parent operations by at least Delay instructions. Because the lower bound on number of registers for a legal canonical order is minReg+⌈Delay/2⌉, this process will require at most ⌈Delay/2⌉ iterations. With this inexpensive extra step, it is possible to further fine-tune DLS to use even fewer registers in many cases when Delay>1.

Delay	DLS Optimal	DLS Sub-Optimal	Total	% DLS Optimal
2	1,015,481	17,930	1,033,411	98.3
3	1,015,481	17,930	1,033,411	98.3
4	1,007,509	25,902	1,033,411	97.5
5	1,007,509	25,902	1,033,411	97.5
6	1,007,511	25,900	1,033,411	97.5
7	1,007,535	25,876	1,033,411	97.5
8	1,007,703	25,708	1,033,411	97.5
9	1,008,631	24,780	1,033,411	97.6

Figure 2.18: Heuristic Results for All Trees of 25 or Fewer Nodes

2.8.3 Empirical Results

The DLS algorithm works extremely well as a heuristic for `Delay` values greater than 1. By enumerating all possible expression trees of 25 or fewer nodes, and testing the algorithm against an exhaustive search algorithm, the effectiveness of the algorithm as a heuristic can be easily verified. We ran trials for `Delay`'s of 2 through 9 to obtain the results given in Figure 2.18. A schedule produced by DLS is considered sub-optimal if it uses R registers but contains no interlocks when there exists a non-interlocking schedule needing fewer than R registers, or if it uses R registers and contains I interlocks when there exists a schedule using R registers that contains fewer than I interlocks. DLS never uses more than `minReg+Delay` registers and produces a non-interlocking schedule for all but a finite number of trees for any given `Delay`.

2.8.4 Anomaly

Using DLS for `Delay`=2, an interesting (and surprising) counter-intuitive result has been found. It is possible for an expression tree to have a sub-tree whose optimal (delay-free) evaluation requires *more* registers than the entire tree's optimal evaluation. The left

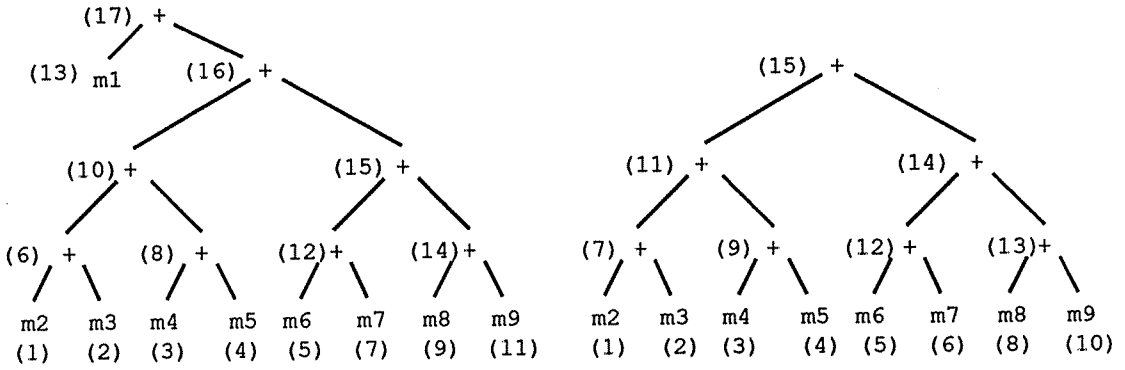


Figure 2.19: Anomaly for `Delay=2`. Entire Tree Needs 5 Registers—Right Sub-tree Alone Needs 6. (The respective optimal evaluation orders are given.)

tree of Figure 2.19 can be evaluated optimally with 5 registers, however its right sub-tree taken alone requires 6 registers for a delay-free evaluation. (No delay-free evaluation exists using fewer than 6 registers, and DLS will find this optimal evaluation.) Note also that the full tree has `minReg=4`, and `Delay=2`, yet it needs only 5 registers for a spill-free, interlock-free evaluation.

2.9 Conclusion

The DLS algorithm presented performs optimal code scheduling and optimal register allocation in linear time for binary expression trees with load delays of 1 cycle. The algorithm can be modified to predict optimal locations for register spilling. Unlike other code scheduling algorithms, it does not suffer from phase-ordering problems with register allocation.

Extensions to the binary tree algorithm allow DLS to optimally schedule trees with non-delayed load instructions and unary operations. DLS can handle (non-shared) register variables optimally. Heuristics for splitting trees and DAGs into forests of

trees allow DLS to schedule entire basic blocks. Basic block scheduling is improved by extending canonical orders to forests of trees that can be merged to fill delay slots.

Furthermore, DLS performs as an excellent heuristic for load delays greater than 1 and can be readily extended to handle tree forests derived from DAGs. As a heuristic it retains its coordination of register allocation and code scheduling without sacrificing its run-time efficiency or conceptual simplicity.

Chapter 3

Probabilistic Register Allocation

Register allocation is the code generation phase that determines which values should be held in machine registers at different points in the program. Most CISC machines (*e.g.*, VAX) can have instruction operands accessed either directly from registers or from memory. The accesses from memory take more cycles than direct register accesses. Most RISC machines cannot access memory except through explicit loads. In either machine class, a value that can be kept in a register rather than memory will be cheaper to manipulate.

Machines, however, have only a very limited number of registers. Modern RISC architectures, like the MIPS R2000 or the SPARC, have fewer than 32 general purpose integer registers available for use. CISC processors typically have even fewer. Therefore, a compiler must often choose between competing values that can be held profitably in registers. Values that are live at a particular point in the program that are not allocated registers must be held in main memory.

Register allocation can be done globally as well as locally. Local allocators make allocation decisions within the limited range of a single basic block. Global allocators

operate at the procedure level—across basic blocks. Often global allocation will allocate a register to a heavily used variable for an entire procedure. To achieve a profitable final allocation, both local and global schemes may choose to keep a particular value in a register at some points in the code, and in memory at others.

For instance, a value may be heavily used in one loop (that will presumably execute often) but may not be used at all in another loop. If registers are scarce, a global allocator may decide to allocate a register to the value only for the loop in which it is heavily used. Not allocating it a register in the other loop will leave the register available to another value that is used in that loop. A local allocator makes similar allocations over different regions of a basic block.

A closely related problem is that of *register assignment.* Register assignment is the problem of determining which actual physical register will hold a particular value (that has previously been *allocated* a register).

3.1 Overview

The dominant paradigm in modern global register allocation is graph coloring ([CH90], [CAC+81], [BCKT89], [LH86]). Unfortunately, graph coloring does not really address the issue of register allocation, but rather the related issue of register assignment. That is, graph coloring tells us how to assign registers so that simultaneously live values aren't assigned the same register. The harder problem—which values to put in *some* register—is not directly addressed. Hence there is always a spilling heuristic that reduces register demand until coloring (register assignment) can succeed.

A famous political maxim states that "All politics is local," and we believe that much the same is true for register allocation. Ultimately, when an operand is actually

used it must be in a register, and once a value is in a register, it is easy to reuse within a basic block. Simple, fast and nearly optimal local register allocators are known [HFG89]. Once local register needs are met, the effects of global allocation can be estimated ([Bea74], [Mor91]). In particular, a good global allocation improves upon good local allocation by eliminating unnecessary loads at the entrance to a basic block and by eliminating unnecessary stores at the exit from a basic block. An initial load of `v` is unnecessary if all predecessors exit with `v` in a register, and a terminal store of `w` is unnecessary if `w` is dead or all succeeding loads of `w` can be replaced with a reference to `w`'s register.

Of course we do not initially know which values will ultimately be allocated to registers, so we take a probabilistic approach. At the basic block level, we know values loaded locally into a register and not overwritten have a 100% probability of exiting in a register. The probability of other values residing in a register on exit depends on their probability of residing in a register upon entrance, the number of unused registers in a block, and the pattern of local register usage. The probability that a value will be in a register upon entrance to a block depends on the probabilities it will exit in a register from all predecessor blocks.

During global register allocation, we will model the competition between register candidates with probabilities. Each instruction that requires a target (destination) register must acquire a register from somewhere—either from a pool of free registers, or, if necessary, from a register candidate. If the instruction must take a register from one of possibly many register candidates, we will assume that it will choose among the competing candidates randomly. For instance, if the instruction must take a register from one of 4 candidates that might be in registers, each of those candidates has a

3/4 probability of "surviving" that instruction. That 3/4 is a measure of the register competition among those candidates at that instruction. By combining probabilities at all instructions where a register candidate is live in a program, we can measure the total competition that candidate faced.

This chapter will describe *probabilistic register allocation.* Our algorithm uses probabilities to measure the competition for registers between register candidates. The algorithm makes allocation decisions driven by estimates of benefit and the measure of competition based on probabilities. Probabilities are used as heuristic measures to drive the register allocator; the register allocator is deterministic.

Once initial estimates of the probability of register allocation are made, these estimates are weighted by the net benefit gained by allocating a given value to a register and the most promising candidate is allocated to a register. Probabilities are recomputed and again the most promising candidate is allocated to a register. This continues until all registers are allocated. The resulting technique is simple and yet identifies those values that can readily and profitably reside in a register.

3.2 Graph Coloring Allocators

The basic graph coloring technique involves creating a register interference graph and then pruning nodes from that graph that can be trivially colored (assigned a physical register) ([CAC+81], [Cha82], [CH90], [LH86], [BCKT89], [CK91]). The nodes of the graph represent the *live ranges* of the different register candidates (variables and temporaries). The live range of a candidate is the set of all program points where that candidate is live—as computed by data-flow analysis. Figure 3.1 gives an example inner-loop of a procedure, and Figure 3.2 gives the interference graph for the variables

and virtual registers.[1] Notice that the graph abstracts away all control-flow information about *how* the different live-ranges interfere with each other.

Given enough registers to "color" all register candidate values, this technique works well. However, once the interference graph is reduced to a graph for which the node pruning heuristic blocks, the allocator must act so that register assignment may continue. Various graph coloring techniques differ precisely in what they do when pruning blocks.

When the pruning heuristic blocks, Chaitin's techniques ([CAC+81], [Cha82]) take the simplest approach. A node (register candidate) is picked based on a cost measure, removed from the graph, and assigned permanently to a memory location. All subsequent references to that value must be from memory.

Priority-based coloring ([CH90], [LH86]) also builds an interference graph and attempts to color it by pruning nodes. If this pruning blocks, a heuristic is employed to split large, costly live ranges into smaller ranges in an attempt to produce a graph that can be further pruned. Complex heuristics are used to split live ranges to minimize the costs of spilling and reloading the values across the boundaries to the new, smaller live-ranges. This heuristic is repeated until all the register candidates are assigned registers.

Callahan and Koblenz allocate registers globally by doing graph coloring "hierarchically" [CK91]. They treat the program as a hierarchy of nested "tiles." Tiles may be basic blocks, conditionals, or loops. They assign registers using graph pruning techniques, but start by assigning registers in innermost tiles and progressively assigning registers in enclosing tiles. This technique succeeds in isolating some local register needs

[1] Only virtual registers `v6` and `v3` are given in the interference graph because the other virtual registers are subsumed by a particular variable.

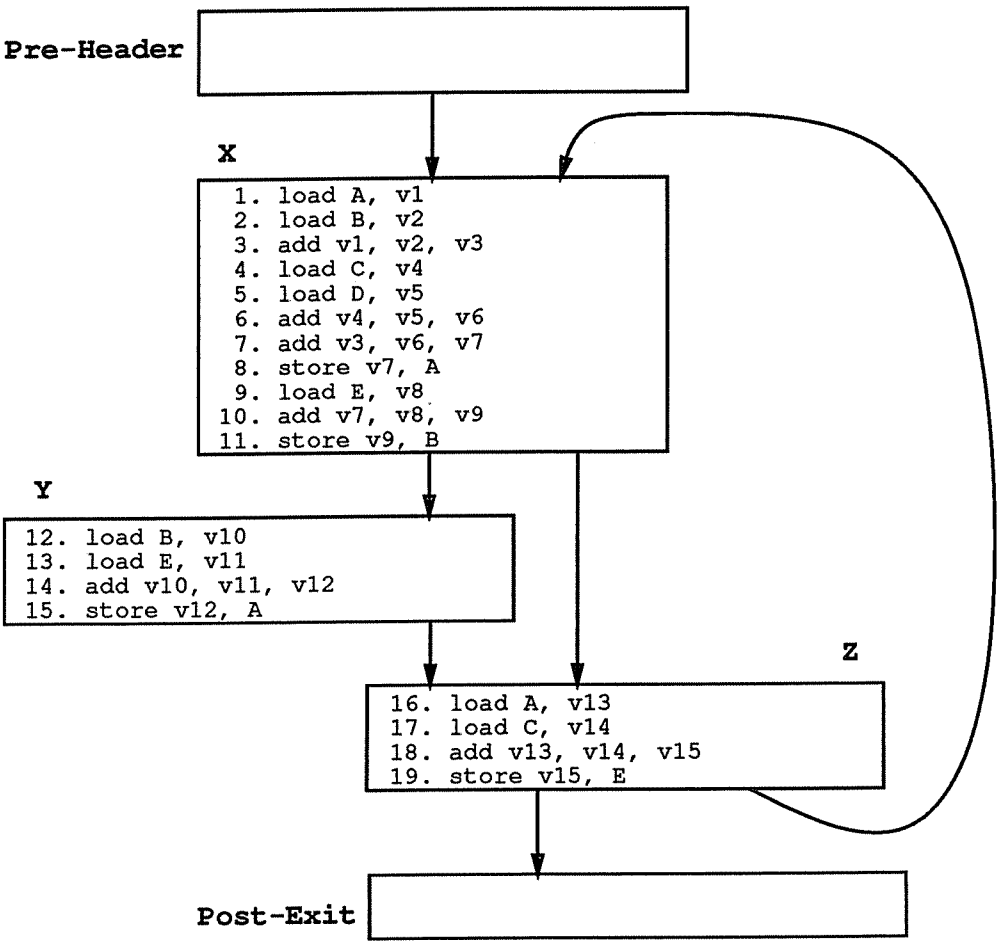


Figure 3.1: Inner Loop

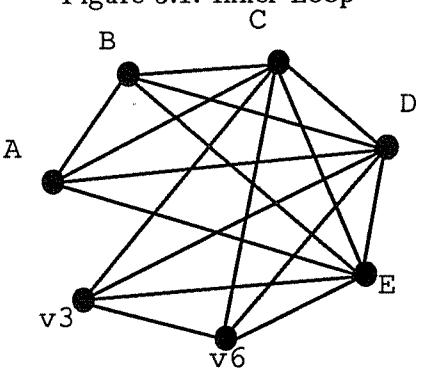


Figure 3.2: Interference Graph

from global register usage so that variables referenced within deeply nested loops will be assigned a register before a variable that is not referenced within the loop. If, however, within a tile, the graph pruning algorithm fails to find a coloring, their technique resorts to the graph-coloring spill techniques outlined in [BCKT89]. Therefore, while succeeding in biasing register allocation within a loop to variables used within that loop, spill decisions must still be made via *ad hoc* heuristic methods.

Few graph coloring techniques do local (basic block level) register allocation as well as established local allocation algorithms ([HFG89], [Fre74], [FL88]). Unlike graph coloring algorithms, local allocation techniques are able to exploit information about the simple sequential nature of register usage in the block to minimize local spill code. This information is lost when register allocation is cast as a graph coloring problem.

3.3 Probabilistic Register Allocation

Our technique, *probabilistic register allocation*, utilizes graph coloring techniques to assign registers, but not to allocate them. Probabilistic register allocation computes probabilities to measure competition between global register candidates so that good allocation decisions are made. Allocation is done prior to assignment based on quantified measures of the costs and benefits of having particular values in registers. Separating allocation from assignment allows our algorithm to concentrate on the important problem of determining which values will profitably be held in registers at different points in the program.

Probabilistic global register allocation follows local allocation. The global allocation proceeds from inner loops to outer loops so that values used within a loop are routinely allocated a register for that loop.

3.3.1 Local Register Allocation and Probabilities

Most local register allocators share the basic principle of deciding what value should stay in a register (or when a spill is necessary) by checking the closeness of the next use of values already in registers ([HFG89], [Fre74], [FL88]). If a value in a register has only distant next uses then it will be spilled before a value to be used sooner. The intuitive explanation for such a heuristic is simple: the farther a potential use of a value is from a program point, the *less likely* the value is to remain in a register all the way to that use, thereby decreasing the expected value of retaining register residence. The likelihood of a value being able to stay register-resident can be viewed as roughly *inversely proportional* to the distance that value would have to stay register-resident. Therefore, heuristics choose to spill the value that seems likely to lose its register anyway. Conversely, one can view retaining a register allocation over a long distance as *more likely* to result in spills of other register values—another motivation for spilling those values with only distant uses.

The terms "less likely" and "inversely proportional" come from probability. To avoid the NP-Complete problems of optimal local register allocation, heuristics use a simpler, *probabilistic* summary of the local circumstances to drive register allocation and spill decisions. We believe that a formal characterization of this summary information as probabilities has been ignored because in straight-line code, distance is always directly related to the probability of either being spilled or causing other values to be spilled. That is, from any given point in a basic block, a distant use has a higher probability of being spilled than a closer use.

3.3.2 Global Register Allocation and Probabilities

The use of probabilistic summary information to drive global register allocation has not been previously studied. Probabilities present the foundation for a global register allocator that combines the advantages of excellent local allocation with effective global allocation. Probabilistic register allocation avoids the problems of live-range splitting that plague graph-coloring techniques [Cha82], [BCKT89], [CH90] by implicitly (and automatically) splitting ranges where the probability and benefit of residing in a register are low.

The probability that a register value will continue to exist in that register at a more distant point in straight-line code can be seen as inversely proportional to the distance to that point. This is not exact, but is intuitively plausible.

Approximating global probabilities requires the ability to handle control flow. Loops and conditionals complicate matters because a value in a register may originate in many different locations and reach many different uses—there is not necessarily a unique next use or a single defining point. Simple data-flow analysis can determine all of the definitions that reach a particular use. The question to answer is "What is the probability that the value will remain in a register over all paths reaching a particular use from all the reaching definitions?"

A simple estimate of the probability can be derived by extending our local, basic block heuristic—count the total number of intervening instructions on all of the paths from definitions to the use in question.

Unfortunately, in practice this yields a relatively poor measure of probability because not all intervening instructions equally affect the probability that a register will need to be spilled. In addition, global register values only need to be spilled when there are

too few registers for both local needs and live global values. If along some path there is a surplus of registers, all variables needing registers along that path can be allocated registers with 100% probability. Likewise, if along some path from a definition to a use the local allocator requires *all* of the registers, it would be impossible for the global value to be in a register along that path and hence the probability of the value being in a register at the use would be 0.

Calculating Local Register Probabilities

The *global* probability that a variable will reside in a register at a given use (load) is computed from the Δ-*probabilities* for that variable in the blocks that reach the load. Δ-probabilities are computed locally (for each live variable) based on local allocations and live variable analysis. Δ-probabilities are computed on a per instruction basis, and indicate the probability that a variable's value will continue to reside in a register after that instruction's register needs are resolved *if* the value had been in a register up to that instruction.

The algorithm maintains a configuration state at each point in the basic block. The configuration is a 4-tuple, (*allocated*, *candidates*, *unallocated*, *possibly*). The values (variables and temporaries) that are allocated to registers make up the set *allocated*. The variables (register candidates) that are competing for registers at a given point in the program are in the set *candidates*. The maximum number of register candidates that could be allocated registers is *possibly*. The count of registers known to contain no useful value is *unallocated*. The total number of allocatable registers (a constant) is *REGISTERS*.

The configuration will always maintain three important invariants. It must be the

case that the sum of the number of allocated registers, possibly allocated registers, and known available registers is equal to the total number of registers available:

$$|allocated| + possibly + unallocated \equiv REGISTERS.$$

The size of each group is bounded below by 0 and above by the total number of registers:

$$0 \leq |allocated|, possibly, unallocated \leq REGISTERS.$$

And, it cannot be the case that there are more candidate values possibly in registers than there are candidates:

$$possibly \leq |candidates|.$$

The Δ-probabilities are calculated after local register allocation by iterating (in order) over the instructions in each basic block. Figure 3.3 gives the algorithm for computing the Δ-probabilities for instructions within a basic block.

Given the local allocation, each instruction will possibly free and allocate registers. Freeing a register will change the configuration by deleting a member of the *allocated* set. If the register held the value of a variable that is still live (determined by global data-flow analysis), then that variable is added to *candidates*, and the *possibly* count is incremented. If the variable is dead, then the *unallocated* count is incremented.

An instruction that allocates a register must acquire a register from either *unallocated* or *possibly*. When $possibly = |candidates|$ and the allocated register will hold a variable from *candidates*, all the Δ-probabilities will be 1, and the register used will come from the set counted by *possibly*. This follows from the fact that when $possibly = |candidates|$, there are exactly enough registers available to hold the elements of *candidates*.

```
1   procedure ComputeDeltas(BasicBlock)
        //Initialize Configuration.
2       allocated ← { variables allocated registers entering the block }
                    // Before global allocation begins, allocated will be empty here.
3       candidates ← { live variables entering BasicBlock } − allocated
4       possibly ← Min(REGISTERS− |allocated|, |candidates|)
5       unallocated ← REGISTERS− (|allocated| + possibly)
6       ∀ insn ∈ BasicBlock do                    // Iterate over instructions in order.
7           ∀ f ∈ registers freed by insn do
8               allocated ← allocated − { f }
9               if f contains the value of v, a live variable then
10                  candidates ← candidates ∪ { v }
11                  possibly ← possibly + 1
12              else
13                  unallocated ← unallocated + 1
14              end if
15          end ∀
16          if insn allocates r, a register then          // Allocate result register.
17              allocated ← allocated ∪ { r }
18              if r holds the value of variable w then
19                  candidates ← candidates − { w }
20              end if
21              if possibly > |candidates| then
                    // Only occurs when possibly = |candidates| prior
                    // to satisfying preceding conditional.
22                  Δ ← 1.0
23                  possibly ← possibly − 1
24              else if unallocated > 0 then
25                  Δ ← 1.0      // Allocating empty register cannot kill anything.
26                  unallocated ← unallocated − 1
27              else
28                  Δ ← (possibly−1)/possibly
29                  possibly ← possibly − 1
30              end if
31          else
32              Δ ← 1.0              // Allocating no register cannot kill anything.
33          end if
34          ∀ v ∈ candidates do
35              ΔTable[insn][v] ← Δ
36          end ∀
37      end ∀
38  end procedure
```

Figure 3.3: Procedure to Compute Δ-Probabilities

Otherwise, if no unallocated registers exist, one of the registers counted by *possibly* must be taken. Remember that *possibly* counts the number of variables in *candidates* that might be allocated registers. Since we may be taking a register from such a variable, we must compute the probability that a particular variable's register will be taken. Given *possibly* registers to choose from, *if* a given variable were in a register, the probability that the allocator would *not* choose that variable's register must be

$$\frac{possibly - 1}{possibly}.$$

Therefore, $(possibly - 1)/possibly$ is the Δ-probability that a live variable (a member of *candidates*) will keep its register past this instruction. If there are unallocated registers, such a register is used first, the *unallocated* count is decremented, and the Δ-probability of all live variables at this instruction is 1. If an instruction does not allocate any registers, it cannot cause any live variables to lose a register, so the Δ-probabilities for all such values is 1.

Example Computation of Local Probabilities

The example in Figure 3.4 illustrates how local allocation, liveness analysis and probabilities interact for potential register variables. (It is based on the program whose flow graph is shown in Figures 3.1 and 3.6.) The example assumes that there are 3 registers available to be allocated among the 5 variables and the intra-block temporary values. After the first instruction, local allocation requires 1 register, so 1 of the 3 registers must be allocated to A at this point, and only 2 of the remaining 3 registers will keep their values on entry. Therefore, the others (B–E) have a 2/3 chance of retaining a register. (We assume that if a value is live on entry, then it *may* be in a register, and we must

Block X

Instr #	Instruction	Local Needs	Δ-Probabilities					Comments
			A	B	C	D	E	
1	load A, v1	1	1*	2/3	2/3	2/3	2/3	
2	load B, v2	2	**1**	1*	1/2	1/2	1/2	
3	add v1, v2, v3	1	0	0	1	1	1	A, B are dead
4	load C, v4	2	-	-	1*	1	1	
5	load D, v5	3	-	-	**1**	1*	0	All registers in use
6	add v4, v5, v6	2	-	-	1/2	1/2	1/2	Reuse v4 or v5
7	add v3, v6, v7	1	1*	-	1	1	1	Reuse v3's register
8	store v7, A	1	**1**	-	1	1	1	
9	load E, v8	2	**1**	-	1	1	1*	Reuse v6's register
10	add v7, v8, v9	1	2/3	1*	2/3	2/3	2/3	
11	store v9, B	0	1	**1**	1	1	1	
Probability Value Remains in Register			2/3	**1**	1/3	1/3	2/3	

Figure 3.4: Δ-Probabilities (3 Registers available for locals and globals)

randomly choose one to give up its register when the local allocator needs an extra register. For simplicity, in the algorithm as implemented, we do not take "*distance to next use*" into account when calculating spill probabilities.) A "1*" indicates that the variable was either loaded or calculated into a register at this instruction and therefore must exist in a register. A bold-face "**1**" indicates that the value has been allocated a register through that instruction. (At this point all allocations are local, but later the global allocation algorithm will also allocate registers into, out of, and through basic blocks for different register candidates.)

The next instruction (#2) requires another register, so one of the remaining variables must be spilled—but not A because the local allocator keeps it in a register for a local use (#3). Because #2 requires 1 of the 2 remaining registers to be spilled, each candidate has a 1/2 chance of retaining a register. Instruction #3 does not lower the probability of any live variable because #3 frees the registers holding both A and B, and both those values are now dead. Therefore, either of those two registers can be used to hold

Block Y

Instr #	Instruction	Local Needs	Δ-Probabilities					Comments
			A	B	C	D	E	
12	load B, v10	1	2/3	1*	2/3	2/3	2/3	
13	load E, v11	2	1/2	**1**	1/2	1/2	1*	
14	add v10, v11, v12	1	1*	**1**	1	1	0	E is dead (global info)
15	store v12, A	0	**1**	**1**	1	1	-	
Probability Value Remains in Register			**1**	**1**	1/3	1/3	0	

Block Z

Instr #	Instruction	Local Needs	Δ-Probabilities					Comments
			A	B	C	D	E	
16	load A, v13	1	1*	2/3	2/3	2/3	2/3	
17	load C, v14	2	**1**	1/2	1*	1/2	1/2	
18	add v13, v14, v15	1	2/3	2/3	2/3	2/3	1*	
19	store v15, E	0	1	1	1	1	**1**	
Probability Value Remains in Register			2/3	2/9	2/3	2/9	**1**	

Figure 3.5: Δ-Probabilities (3 Registers available for locals and globals)

the result of the addition without having to spill any other register candidates. It is impossible (probability = 0) for E to be register-resident at #5 because the instruction sequence #1–5 requires all 3 registers.

The bottom row of the table in Figure 3.4 indicates the probability that a variable will be in a register on exit. These values were calculated by multiplying together all the Δ-probabilities of the variable from the last point it was certain to be in a register to the end of the block.

It is also possible to calculate the *conditional probabilities* for variables that are not referenced within a basic block. If such a variable is in a register on entry to a block, the product of the Δ-probabilities for the entire block is the probability that it will be in a register on exit.

Figure 3.5 gives two example basic blocks with such "pass-through" values. In the top example (**Block Y**), the exit probabilities for A, B, and E are absolute probabilities

(because the variables are referenced within the block). The probabilities for C and D are conditional—they are the probability that the value will still be in a register on exit if it was in a register on entry.

The computation of the Δ-probabilities at #14 (Figures 3.5 and 3.6) demonstrates the powerful interaction of local allocation and global data-flow analysis. Global data-flow analysis indicates that the value of E is dead after #14, and, therefore, the local allocator may reallocate its register to hold the value of the computation (A). Because no additional registers were needed by the local allocator, the Δ-probabilities of B, C, and D are 1, reflecting no competition for registers (at that instruction).

Calculating Global Register Probabilities

Given the Δ-probabilities for each variable at each instruction, it is a simple matter to determine the probability that a variable will be available in a register at a particular load. Data-flow analysis isolates all the reaching *register-definitions* of the variable. A register-definition is any point in the program that the variable is known to be register-resident due to previous allocation decisions. The **Pre-Header** is also considered to be a register-definition of *every* variable, since loads can be added there as needed. A register-definition, d, reaches a load, ℓ, if there exists a register-definition-free path from d to ℓ. The set of all program points along such definition-free paths is the *register-live-range* of the load. To remove the load of a variable, the variable must be allocated a register over its entire register-live-range.

The Δ-probability of a particular instruction is the probability that a variable's value will continue to be in a register after that instruction executes if it were in a register before the instruction. Therefore, the probability that none of the instructions in the

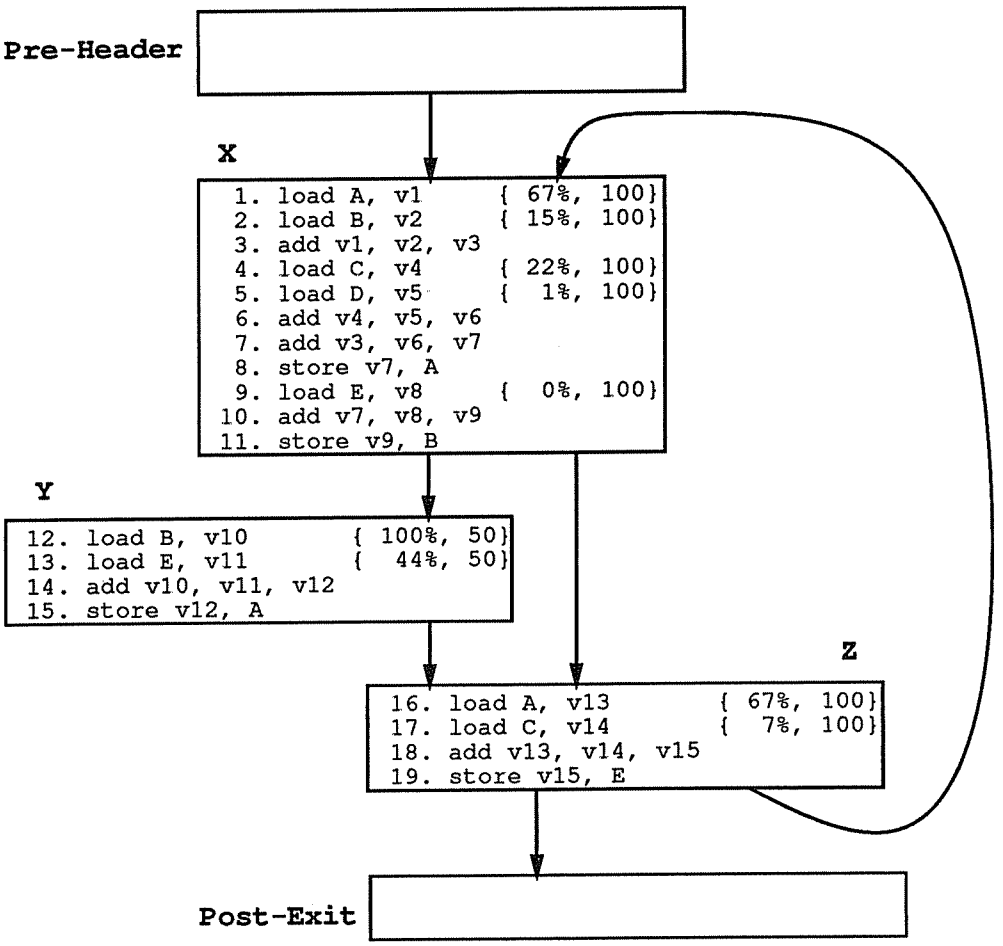


Figure 3.6: Sample Inner Loop with Global Probabilities (Assuming 3 Registers). The loads are annotated with $\{probability, benefit\}$ pairs.

register-live-range will kill the value is simply the product of all their Δ-probabilities.

The computation of global probabilities is done over the entire register-live-range rather than on a per path basis because a load cannot be removed unless the variable is allocated a register along *all* paths from (possibly many) reaching register-definitions to the load. The register-live-range is precisely the set of all such program points.

The inner loop in Figure 3.6 has been annotated with global probabilities of register

residence at each load assuming there are 3 registers available. (Blocks **X**, **Y**, and **Z** are those given in Figures 3.4 and 3.5.) For instance, the probability of C being in a register at #4 is equal to the product of the Δ-probabilities for #18, #19, #1, #2, and #3. (There are two register-definitions that reach the load at #4: one in the **Pre-Header**, and one at #17.) The probability is, therefore,

$$2/3 \times 1 \times 2/3 \times 1/2 \times 1 = 2/9 \approx 22\%$$

Similarly, to remove the load of A at #16, A must be register resident from both of its reaching register-definitions at #10 and #15. (Instruction #10 is a register definition because A is in v7, an operand of #10's addition.) Therefore, the register-live-range of A's load at #16 is #10, #11, and #15, with Δ-probabilities of 2/3, 1, and 1, respectively. The probability of A being register-resident at #16 is, therefore, 2/3 (the product of the Δ-probabilities).

The load of C at #4 has a greater global probability than the load of B at #2 even though it follows #2 in the basic block. This can be explained intuitively by observing that B must be register-resident through more of the loop (#13–19, #1) than C (#18–19, #1–3).

The extremely low probability (1%) for the "`load D, v5`" at #5 is a consequence of the fact that for D to be available in a register at that location, it must be in a register for the entire loop.

3.4 Probabilities Guide Global Register Allocation

Our global register allocation algorithm uses local probabilities combined with a measure of benefit to determine which variable uses should be globally allocated registers. Uses with large benefit and high probability are given the greatest register allocation priority. We combine the measure of benefit and probabilities by multiplying them together to get a *figure of merit.* Candidates with a higher figure of merit will be given priority.

The benefit of allocating a register to a use (and hence to all the paths reaching that use) is determined by estimating how often the use will be executed, and hence how many cycles may be saved if the value is accessed from a register rather than from memory. This can be determined heuristically from loop and conditional nesting levels, or empirically through profiling information from previous executions of the program [BL92].

Once a particular use has been allocated a register, there is no need to do a load of the variable at that use, thus saving time and space. Allocating a register causes the probabilities for other inter-block variables to change. This follows from the observation that if a register becomes allocated at some point in the program, it must have a probability of 1 at that point. To make room for this fixed probability (1), the probabilities of any competing candidates must change.

If, for example, `B` and `E` in block Z become allocated, then after the load of `A` at #16 all three of the registers will be in use. Therefore the probability that `C` could be in a register at #17 must drop to 0. Because `E` is dead at #17, its register will be allocated to `C` when `C` is loaded.

Because probabilities and allocations interact, global allocation is done iteratively. A greedy algorithm finds the best candidate for register allocation based on their relative

figures of merit (*probability* $\times$ *benefit*). Then the probabilities are recalculated, and another register is allocated. This process is repeated until there are no remaining uses with probability greater than 0.

Recomputing Δ-probabilities requires that *ComputeDeltas*() (Figure 3.3) be re-executed for each block in which global allocations have taken place. These global allocations change the set of registers that are allocated upon entry, *allocated*, and they change which registers are freed by various instructions—allocated registers are no longer freed.

3.4.1 Improving Probabilistic Register Allocation

Probabilistic register allocation carefully determines which candidates for registers show the greatest promise to benefit the program. Additional techniques complete the register allocation process. Our algorithm allocates registers *inside-out*, from the most deeply nested regions to the outermost, thereby emphasizing allocation in the code most likely to be executed often. The algorithm assumes that all variables initially reside in memory until they are allocated a register. After local allocation, the algorithm locates loads that can be removed by allocating a register along all paths from reaching definitions to the chosen load. After loads are removed, it is possible that some stores may be removed—those whose values are no longer referenced from memory.[2] In addition, some loads and stores may be removed from a loop by placing them outside the loop in a pre-header or post-exit to increase loop speed. Local allocation, followed by load and store removal, provides a mechanism for global allocation that avoids the difficulties of splitting live-ranges or isolating spill candidates.

[2]Care must be taken not to remove the final store of a global variable that may be accessed by other procedures.

3.4.2 Example

The example in Figure 3.6 illustrates a simplified version of our algorithm (again, assuming 3 registers). Loads will be examined for removal in order of benefit and probability. The second number of each tuple is the *benefit* of removing the load—a value of 100 indicates the load will be executed on all of the 100 iterations of the loop whereas a value of 50 represents an estimate that that branch of the conditional will only execute half as often. (We assume that the probability that a value resides in a register from the **Pre-Header** is 100% because that can be made so by adding a load of the value there.)

Because #12 has a 100% probability of finding `B` in a register, its consideration could be deferred indefinitely—it is certain to be in a register! This 100% chance is easily deduced by recognizing that block **Y** has only **X** as a predecessor, and #10 (in **X**) left `B` in a register. (This is information that an interference graph could not readily provide.) Delaying allocation (and the subsequent removal of the load) would, however, contribute spurious Δ-probabilities to the calculation of global probabilities for other uses; we therefore remove it immediately. The (possibly) inaccurate Δ-probabilities occur because the load allocates a register for its result, and that may lower probabilities for competing register candidates. Delaying the removal could temporarily lower those probabilities and cause the global allocator to discriminate against those candidates.[3]

The loads of `A` at #1 and #16 will be removed next because they have the highest probabilities (67%) of those loads with a benefit of 100 (and hence the highest "merit" value of 67). In order to remove the load at #1, it is necessary to ensure that `A`'s value

[3]All loads with 100% probability of register allocation are removed immediately, regardless of their benefit. Only this situation of 100% probability (*certainty*) is treated as a special-case and it is done to ensure accurate Δ-probability computations.

will be in a register on the incoming control-flow arc from the **Pre-Header** by putting a load of A there.

After registers have been allocated in order to remove these three loads (#1, #12, #16), the probabilities for the remaining loads are recalculated. The new Δ-probabilities are given in Figure 3.7, and the new global probabilities (calculated using these new Δ's) are given in Figure 3.8.

The Δ-probabilities for block **Y** of Figure 3.7 have undergone a dramatic change after the allocation of registers for A and B. Because A is allocated a register upon entry, and because A is dead at #13, the register allocator can reuse that register for the load of E. Therefore, the Δ-probabilities for C and D are 1 throughout the block. The summary "*Probability Value Remains in Register*" seems to indicate that 4(!) values have a 100% probability of being in a register at the end of the block. This result can be explained by recognizing that the probabilities for C and D are conditional—the value of C (or D) is in a register on exit *only if* it is in a register on entry. Since at most one of the two could have been in a register upon entry, we have not erroneously calculated that 4 values could fit in 3 registers.

Now, four remaining loads have the highest "figure of merit" (25) for removal: three loads with benefit of 100 and a probability of 25% (#2(B), #4(C), and #17(C)), and one load with benefit of 50 and a probability of 50% (#13(E)). We will arbitrarily choose to allocate a register to C at #4—this too requires an initial load of C in the **Pre-Header**. After this allocation, the remaining probabilities change again—only two of the remaining load instructions have probability greater than 0: #13 and #17. Fortunately, both of these loads can be removed.

It is useful to examine how probabilities could determine that D could not be in a

Block X

Instr #	Instruction	Local Needs	Δ-Probabilities					Comments
			A	B	C	D	E	
1	-	0	**1**	-	-	-	-	*Removed*
2	load B, v2	1	**1**	1*	1/2	1/2	1/2	
3	add v100, v2, v3	1	0	**1**	1	1	1	A is dead
4	load C, v4	2	-	0	1*	1	1	B is dead
5	load D, v5	3	-	-	**1**	1*	0	
6	add v4, v5, v6	2	-	-	1/2	1/2	-	
7	add v3, v6, v100	1	1*	-	1	1	-	
8	store v100, A	1	**1**	-	1	1	-	
9	load E, v8	2	**1**	-	1	1	1*	
10	add v100, v8, v200	1	**1**	1*	1/2	1/2	1/2	
11	store v200, B	0	**1**	**1**	1	1	1	
Probability Value Remains in Register			**1**	**1**	1/4	1/4	1/2	

Block Y

Instr #	Instruction	Local Needs	Δ-Probabilities					Comments
			A	B	C	D	E	
12	-	0	**1**	**1**	-	-	-	*Removed*
13	load E, v11	1	0	**1**	1	1	1*	A is dead
14	add v200, v11, v100	0	1*	**1**	1	1	0	E is dead
15	store v100, A	0	**1**	**1**	1	1	-	
Probability Value Remains in Register			**1**	**1**	1	1	0	

Block Z

Instr #	Instruction	Local Needs	Δ-Probabilities					Comments
			A	B	C	D	E	
16	-	0	**1**	-	-	-	-	*Removed*
17	load C, v14	1	**1**	1/2	1*	1/2	1/2	
18	add v100, v14, v15	1	**1**	1/2	1/2	1/2	1*	
19	store v15, E	0	**1**	1	1	1	**1**	
Probability Value Remains in Register			**1**	1/4	1/2	1/4	**1**	

Figure 3.7: Δ-Probabilities after 2 Allocations for A (v100), and 1 allocation for B (v200).

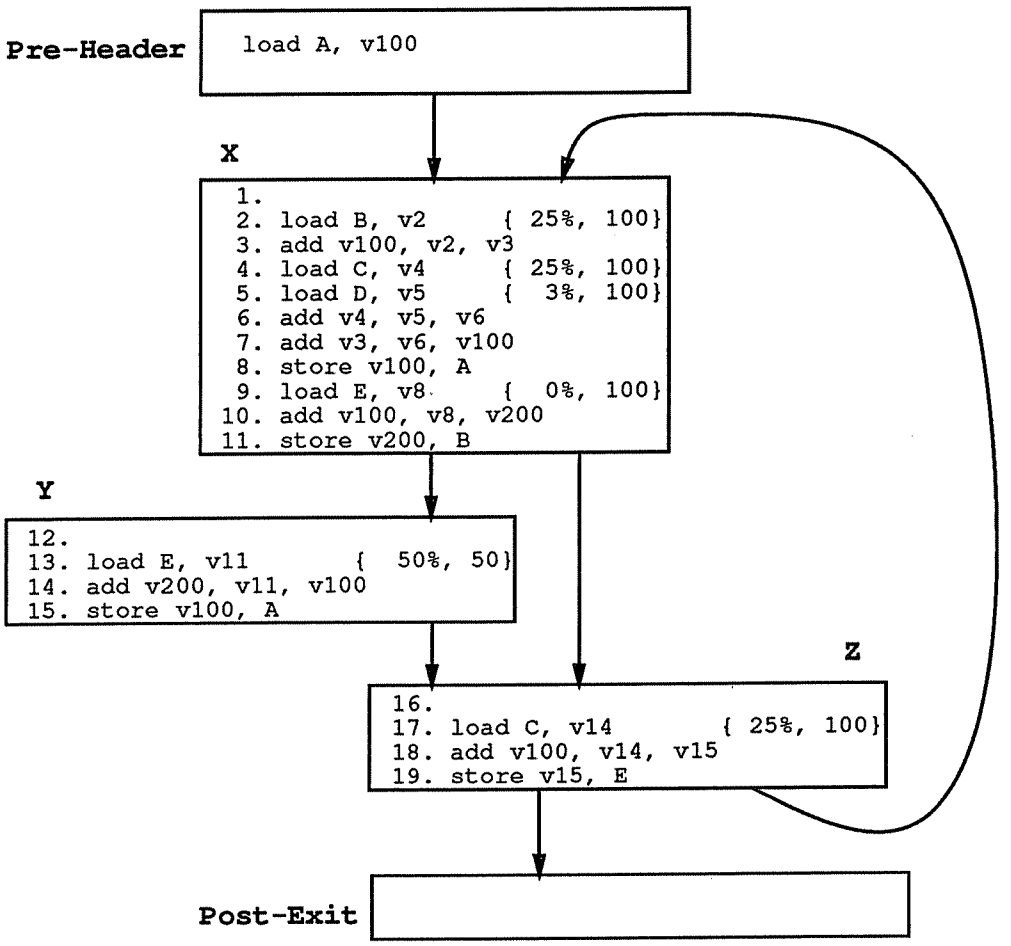


Figure 3.8: Global Probabilities after 3 Allocations

register at #5 after removing the load of C at #4. It might appear that only 2 of the 3 registers are allocated at the entrance to block X (for registers A and C), but prior to loading D it is necessary to load B without destroying A or C and this uses all three registers. Similarly, B cannot be in a register at #2 because all three registers must be in use at the end of block Z holding the values of A, C, and E thus preventing the value of B from being held in a register over that control flow path.

After the loads are removed, simple data-flow analysis indicates that further improvements can be made by removing the stores of A (at #8 and #15). If A is live on exit to the loop, a store of A must be added to the **Post-Exit**.

Note that removal of the "load B, v10" at #12 was possible *without* allocating a register to B for its entire live-range. B must be stored at #11 so that it may be loaded at #2 in a subsequent iteration, **but** probability analysis indicated that the value stored at #11 would be available at #12 in a register, so the load could be removed.

After register allocation, A and C were effectively allocated registers for the inner region, and a load of B was removed. In total, 5 of 8 loads and 2 of 4 stores were removed from the loop. Figure 3.9 gives the code as it would appear after allocation and assignment with the register contents after each instruction given in parenthesis.

By chance, our implementation actually gives a better allocation than the one described above by removing instructions #1, 2, 8, 11, 12, 13, 15, and 16 for an estimated cost of 50 fewer instruction. The better allocation was found because the implementation arbitrarily chose to remove instruction #2 whereas the previous (hand-calculated) example chose #4. The implementation simply happened to break the tie between the candidates of merit 25 differently.

3.4.3 Probabilities Improve Beatty's Algorithm

Our algorithm is an improvement to Beatty's register allocation scheme [Bea74]. His algorithm does local allocation followed by global allocation through the removal of loads and stores to loop pre-headers and post-exits. Our algorithm differs from his in two important ways: ours uses probabilities to provide better global allocation, and ours separates register assignment from register allocation. Beatty's algorithm uses only

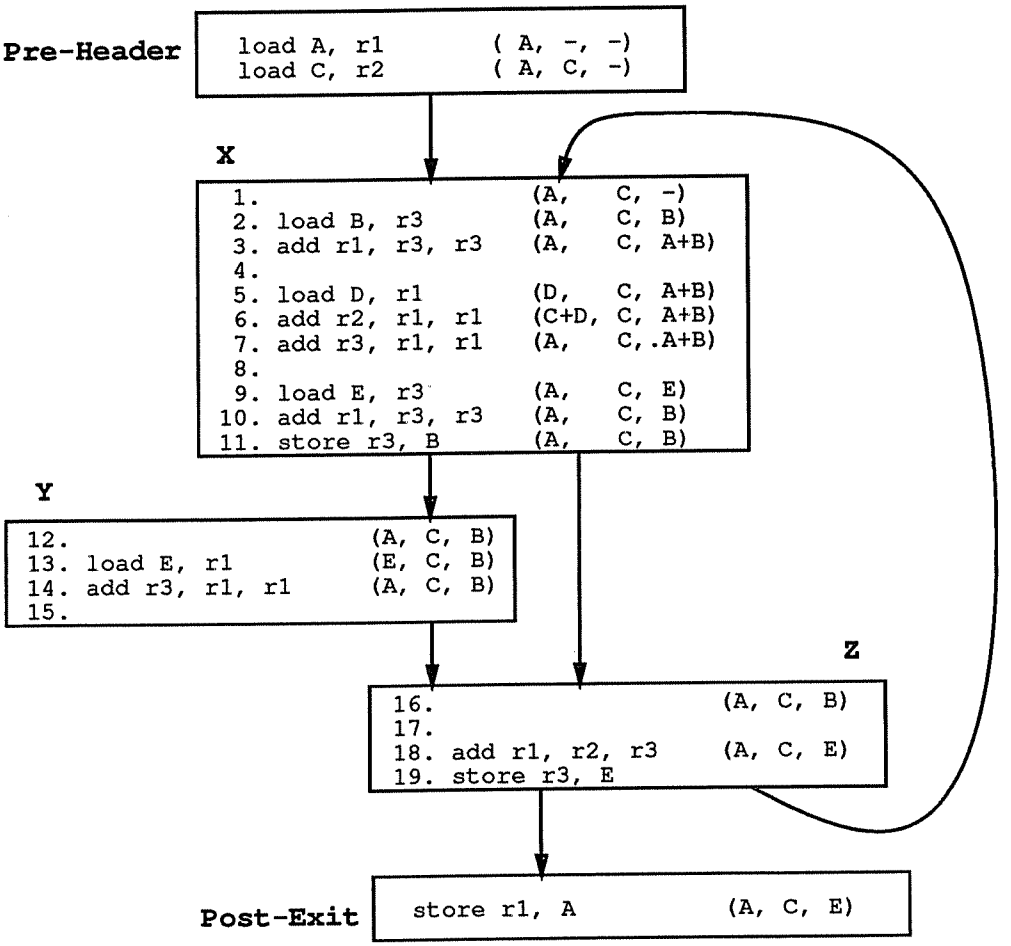


Figure 3.9: Code After Register Allocation and Assignment (Register Assignments in Parenthesis)

benefit analysis (estimates of execution count) to determine which global entities are good candidates for register allocation. Absolutely no attempt is made to quantify the effects of allocating a register to a particular set of paths leading to a use. Probabilities directly measure the *costs* of allocating a register in a particular code region; a low probability indicates there is great competition for registers in the region, a high probability indicates less competition. (A probability of 1 indicates absolutely no competition, and

hence a *free* allocation.) Probabilistic cost measures improve Beatty's algorithm by guiding it to make good allocation decisions by balancing benefit **and** cost estimates.

Our probabilistic register allocator works in three phases (in order): local register allocation, global register allocation, and register assignment. Local allocation is done by the simple, effective scheme of spilling (only when necessary) that value whose next use is most distant. Global allocation is done as outlined in the previous sections. Beatty's algorithm combines global allocation and register assignment into one phase (much as graph coloring algorithms do). Keeping allocation separate from assignment simplifies and improves the register allocation/assignments that can be found.

3.4.4 Register Assignment

Once registers have been allocated to live-ranges, it is necessary to assign registers to them. The previous allocation phases guarantee that there will never be a point in the program that is over-allocated, but so far no legal assignment has been found. All of the variables that have been allocated registers are assigned registers using graph-coloring techniques [Cha82], [BCKT89], [CH90]. An important difference between using graph-coloring for allocation (as other algorithms do) and for assignment (as we do) is that failure to find a legal coloring (unlikely) does not necessitate spilling a value.

It is possible that a program that is not over-allocated may not have a legal assignment. This occurs when the pattern of interferences between the register candidates is such that while there are never more than N candidates live at any point in the program, an N-coloring does not exist for the entire program. To create a legal assignment in such a situation, it would be necessary to either insert code to change register assignments of one or more candidates at different points in the program, or duplicate a code fragment

with a new set of register assignments that avoid the conflicts.

Fortunately, situations with legal allocations, but no legal assignment arise infrequently. In the testing of this register allocation technique, this occurred only for the SPEC89 benchmark, nasa, when it was compiled with only 6 integer registers available.

3.5 Implementation Results

A prototype probabilistic register allocator has been built as part of an experimental code generator for an ANSI C compiler ("lcc" [FH91b] [FH91a]). The code generator produces MIPS R2000 assembler.

3.5.1 Stanford Benchmarks

The tables in Figure 3.10–3.11 summarizes the results of running the compiler on the Stanford benchmarks suite. Each program was run with three different register configurations for both integer and floating point registers. For integer registers the configurations were 19 registers (9 caller-saved, 10 callee-saved), 12 registers (6, 6), and 6 registers (3, 3). For floating point, the configurations were 11 registers (5, 6), 8 registers (4, 4), and 4 registers (2, 2). The numbers of loads and stores in the table represent *dynamic* execution counts. The columns labeled *Total* represent the actual number of loads and stores that could possibly have been removed by register allocation had there been an unlimited number of registers.[4]

The results do not include the cost of saving and restoring callee-saved registers.

[4]For a few programs (*e.g.*, intmm.c), the number of possible loads that may be removed is *overstated* because the statistics do not reflect the fact that an initial load of a parameter passed on the stack is unavoidable. This, of course, results in an *understatement* of our algorithm's effectiveness.

		Load Removal (Execution Counts)			
Program	Registers (integer)	Total	Locally Removed	Globally Removed	Percent
intmm.c	19	977854	521726	454528	99.8
	12	977854	521726	454528	99.8
	6	977854	457726	200128	67.3
queens.c	19	696950	220450	465200	98.4
	12	696950	220450	465200	98.4
	6	696950	220450	288100	73.0
quick.c	19	734725	273098	461627	100.0
	12	734725	273098	461627	100.0
	6	734725	257468	414742	91.5
towers.c	19	614192	458887	138908	97.3
	12	614192	458887	138908	97.3
	6	614192	450696	138908	96.0
fft.c	19	2036794	1598413	438221	99.9
	12	2036794	1598413	392001	97.7
	6	2036794	1352023	303859	81.3
	(floating)				
fft.c	11	11090	5170	5895	99.8
	8	11090	5170	5895	99.8
	4	11090	5170	5850	99.6

Figure 3.10: Read Removal Results on Selected Stanford Benchmarks.

This cost is negligible in all but the recursive procedures (*e.g.*, towers.c and puzzle.c), and is modest and essentially unavoidable in those.

The great number of locally removed loads and stores is the result of the fact that all basic block level temporaries that are used more than once are treated like local variables—via a single initializing store followed by subsequent loads. This mechanism creates a many such "variables" whose stores/loads are almost always removed subsequently by the local register allocator.

The Stanford Benchmarks that are not listed all had 100% of the possible loads removed at *all* register levels. The sub-optimal (and identical) results for intmm.c,

		Store Removal (Execution Counts)			
Program	Registers (integer)	Total	Locally Removed	Globally Removed	Percent
intmm.c	19	272247	192002	80245	100.0
	12	272247	192002	80245	100.0
	6	272247	128002	75445	74.7
queens.c	19	183401	5650	177751	100.0
	12	183401	5650	177751	100.0
	6	183401	5650	155151	87.7
quick.c	19	213785	101453	112332	100.0
	12	213785	101453	112332	100.0
	6	213785	101453	58032	74.6
towers.c	19	311413	180258	131155	100.0
	12	311413	180258	131155	100.0
	6	311413	180258	114772	94.7
fft.c	19	479938	401267	78491	99.9
	12	479938	401267	73191	98.9
	6	479938	360181	67849	89.2
	(floating)				
fft.c	11	1496	1024	446	98.3
	8	1496	1024	446	98.3
	4	1496	1024	425	96.7

Figure 3.11: Write Removal Results on Selected Stanford Benchmarks.

queens.c, and fft.c (floating) at the two greatest register levels are due to charging for the unavoidable loads of parameters passed on the stack—100% of the other loads were removed at both register levels. The identical (sub-optimal) results for towers.c at 12 and 19 registers stem from an implementation bug that created a load of a local variable at the beginning of a procedure (its *pre-header*) because a definition-free path existed from the beginning of that procedure to the potentially uninitialized use of that variable. Except for this additional load, all other possible loads were removed from towers.c at 12 and 19 registers.

3.5.2 SPEC Benchmarks

Our algorithm removed virtually all of the loads/stores in the Stanford Benchmarks. We also tested the system on the SPEC89 Benchmarks.[5] We used `f2c`, a FORTRAN-to-C conversion utility, to convert the FORTRAN benchmarks to C. Figures 3.12–3.15 give the results for load and store removal.

For floating point loads and stores, only `nasa`, `doduc`, `tomcatv`, and `fpppp` (all FORTRAN benchmarks) showed marked degradation with only 4 registers. The table indicates that the entire decrease is attributable to local allocation. The extremely low number of floating point loads/stores done by `matrix300` (a floating point matrix multiplication routine) resulted from the fact that `lcc` does not do any analysis to create register candidates from globally declared variables, and `matrix300` does not have very many local variables. (The spurious *addition* of floating point loads for `li` resulted from the same bug that introduced loads into towers.c.)

With 6 integer registers, the `nasa` benchmark failed to compile because the legal allocation found by the probabilistic algorithm was not colorable via our coloring heuristic. It did compile with only 5 registers.

Only `nasa` and `matrix300` degraded significantly when integer registers were limited to the lowest level (5 and 6, respectively). For `nasa` this resulted from both local and global pressure. For `matrix300`, this is caused by inner-loops that reference many scalar variables that hold index values or array addresses. With only 6 registers, some of those must be accessed from memory.

[5] We excluded the GNU C compiler because it would not compile under ANSI C.

Program	Registers (Float)	Load Removal (Execution Counts) Total	Locally Removed	Globally Removed	Percent
008.espresso	11	4,484	2,832	1,652	100.0
	8	4,484	2,832	1,652	100.0
	4	4,484	2,832	1,652	100.0
013.spice2g6	11	331,605,625	203,276,714	128,328,911	100.0
	8	331,605,625	203,276,714	128,328,911	100.0
	4	331,605,625	179,883,961	128,328,911	92.9
015.doduc	11	141,749,098	131,183,315	8,283,957	98.3
	8	141,749,098	126,550,890	8,283,957	95.1
	4	141,749,098	110,249,145	8,283,957	83.6
020.nasa7	11	324,916,408	320,784,101	4,132,307	100.0
	8	324,916,408	320,784,101	4,132,307	100.0
	4	324,916,408	268,421,301	4,132,307	83.8
022.li	11	0	0	-1,496,311	N/A
	8	0	0	-1,496,311	N/A
	4	0	0	-1,496,311	N/A
023.eqntott	11	0	0	0	N/A
	8	0	0	0	N/A
	4	0	0	0	N/A
030.matrix300	11	6	3	3	100.0
	8	6	3	3	100.0
	4	6	3	3	100.0
042.fpppp	11	25,824,867	24,333,112	1,491,755	100.0
	8	25,824,867	23,680,664	1,491,755	97.4
	4	25,824,867	20,412,080	1,491,755	84.8
047.tomcatv	11	195,075,700	156,060,400	39,015,300	100.0
	8	195,075,700	156,060,400	39,015,300	100.0
	4	195,075,700	136,552,900	39,015,300	90.0

Figure 3.12: SPEC Benchmark Results: Float Loads Removed.

		Store Removal (Execution Counts)			
Program	Registers (Float)	Total	Locally Removed	Globally Removed	Percent
008.espresso	11	4,762	1,350	3,412	100.0
	8	4,762	1,350	3,412	100.0
	4	4,762	1,350	3,412	100.0
013.spice2g6	11	186,983,134	58,409,740	128,573,394	100.0
	8	186,983,134	58,409,740	128,573,394	100.0
	4	186,983,134	39,748,826	128,573,394	90.0
015.doduc	11	53,353,887	43,435,470	8,462,073	97.2
	8	53,353,887	41,070,951	8,462,073	92.8
	4	53,353,887	32,911,291	8,462,073	77.5
020.nasa7	11	156,163,274	152,023,017	4,140,257	100.0
	8	156,163,274	152,023,017	4,140,257	100.0
	4	156,163,274	99,660,217	4,140,257	66.4
022.li	11	0	0	0	N/A
	8	0	0	0	N/A
	4	0	0	0	N/A
023.eqntott	11	0	0	0	N/A
	8	0	0	0	N/A
	4	0	0	0	N/A
030.matrix300	11	4	1	3	100.0
	8	4	1	3	100.0
	4	4	1	3	100.0
042.fpppp	11	10,693,636	9,201,881	1,491,755	100.0
	8	10,693,636	8,875,657	1,491,755	96.9
	4	10,693,636	6,585,745	1,491,755	75.5
047.tomcatv	11	91,035,400	52,020,100	39,015,300	100.0
	8	91,035,400	52,020,100	39,015,300	100.0
	4	91,035,400	32,512,600	39,015,300	78.5

Figure 3.13: SPEC Benchmark Results: Float Stores Removed.

		Load Removal (Execution Counts)			
Program	Registers (Integer)	Total	Locally Removed	Globally Removed	Percent
008.espresso	19	1,698,143,794	709,926,046	988,180,477	99.9
	12	1,698,143,794	709,926,046	987,671,146	99.9
	6	1,698,143,794	709,925,842	817,601,836	89.9
013.spice2g6	19	1,227,321,939	966,962,203	257,639,982	99.7
	12	1,227,321,939	966,962,203	257,639,982	99.7
	6	1,227,321,939	966,962,203	257,602,630	99.7
015.doduc	19	238,846,530	160,806,536	70,652,862	96.9
	12	238,846,530	160,806,536	70,310,650	96.7
	6	238,846,530	160,806,536	68,860,359	96.1
020.nasa7	19	8,761,160,769	7,264,062,350	1,497,087,513	99.9
	12	8,761,160,769	7,264,062,350	1,101,368,913	95.4
	5	8,761,160,769	6,019,475,400	335,604,296	72.5
022.li	19	1,952,143,913	849,876,710	1,091,165,086	99.4
	12	1,952,143,913	849,876,710	1,091,165,086	99.4
	6	1,952,143,913	849,876,710	1,086,532,001	99.1
023.eqntott	19	777,044,072	241,304,263	535,724,423	99.9
	12	777,044,072	241,304,263	535,724,423	99.9
	6	777,044,072	241,304,263	533,161,877	99.6
030.matrix300	19	2,178,363,191	870,507,685	1,306,401,029	99.9
	12	2,178,363,191	870,507,685	1,304,231,427	99.8
	6	2,178,363,191	870,507,685	869,331,632	79.8
042.fpppp	19	21,166,919	12,447,767	5,981,691	87.0
	12	21,166,919	12,447,767	5,905,762	86.7
	6	21,166,919	12,429,231	5,761,920	85.9
047.tomcatv	19	676,717,877	643,949,977	32,767,900	100.0
	12	676,717,877	643,949,977	32,767,900	100.0
	6	676,717,877	630,944,977	26,214,300	97.1

Figure 3.14: SPEC Benchmark Results: Integer Loads Removed.

		Store Removal (Execution Counts)			
Program	Registers (Integer)	Total	Locally Removed	Globally Removed	Percent
008.espresso	19	640,130,571	288,578,937	351,546,310	99.9
	12	640,130,571	288,578,937	350,604,645	99.8
	6	640,130,571	288,578,916	267,181,168	86.8
013.spice2g6	19	519,393,524	383,476,762	135,916,762	100.0
	12	519,393,524	383,476,762	135,916,762	100.0
	6	519,393,524	383,476,762	135,889,622	99.9
015.doduc	19	80,001,256	66,660,868	13,109,388	99.7
	12	80,001,256	66,660,868	13,037,874	99.6
	6	80,001,256	66,660,868	12,064,351	98.4
020.nasa7	19	2,763,958,479	2,759,472,520	4,485,059	99.9
	12	2,763,958,479	2,759,472,520	4,200,557	99.9
	5	2,763,958,479	2,022,665,570	2,569,471	73.2
022.li	19	975,099,481	312,081,752	656,697,223	99.3
	12	975,099,481	312,081,752	656,697,223	99.3
	6	975,099,481	312,081,752	632,403,818	96.8
023.eqntott	19	456,491,641	81,998,573	374,439,604	99.9
	12	456,491,641	81,998,573	374,439,604	99.9
	6	456,491,641	81,998,573	371,971,351	99.4
030.matrix300	19	440,684,228	435,612,636	5,071,568	99.9
	12	440,684,228	435,612,636	5,066,744	99.9
	6	440,684,228	435,612,636	3,612,334	99.6
042.fpppp	19	6,859,283	4,706,221	1,929,264	96.7
	12	6,859,283	4,706,221	1,882,923	96.0
	6	6,859,283	4,687,685	1,686,177	92.9
047.tomcatv	19	266,895,460	266,767,360	128,100	100.0
	12	266,895,460	266,767,360	128,100	100.0
	6	266,895,460	253,762,360	102,500	95.1

Figure 3.15: SPEC Benchmark Results: Integer Stores Removed.

3.5.3 Comments

Unfortunately, `lcc` does not do any global optimizations such as alias analysis or global common subexpression elimination that would create many additional candidates for global register allocation and, therefore, test our techniques more strenuously. (`Lcc` only considers temporaries, and scalar local variables and parameters as candidates for register allocation.) We have, therefore, chosen to create a similar register scarcity artificially by lowering the number of available registers. While not a perfect measure of how the algorithm would do in an optimizing compiler, the numbers are impressive nonetheless.

3.6 Compiler Performance

The current register allocator is a prototype implementation that slows `lcc`'s compilation rate from over 1000 lines/sec. to about 50 lines/sec. The naive implementation recomputes data-flow and Δ-probability information for the current loop after each load is removed. Doing this recomputation incrementally would likely increase the allocation speed considerably.

Alternatively, a quicker, but less accurate, allocation heuristic could be built that would not recompute these values, but would instead use a static "snapshot" of the probabilities. This would avoid recomputing the Δ-probabilities after each load is removed. The static initial probabilities would still serve as an accurate metric of the relative scarcity of registers faced by each register candidate.[6]

[6]The current algorithm terminates when all the global probabilities are 0. If Δ-probabilities are not recalculated, it would be necessary to terminate when it is determined that no more candidates can fit in the available registers.

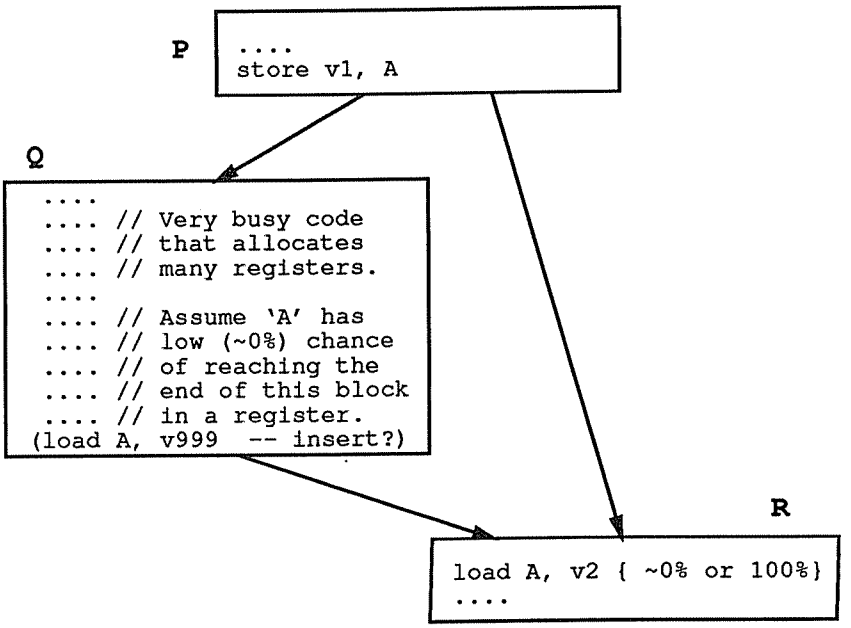


Figure 3.16: Inserting a Helpful Load

3.7 Algorithm Extensions

3.7.1 Manipulating Probabilities

Probabilities are not cast in stone. As the previous algorithm demonstrates, the probabilities of different values being allocated a register change as the program is transformed. With each removal of a load or store, some of the remaining values had changes to their probabilities.

It is possible—and in some cases desirable—to artificially manipulate probabilities by transforming the program. For instance, inserting a load of a particular value on a path to a second load of that value will almost certainly increase the probability of that value being in a register at the original load. In fact, the *closer* the inserted load is to the latter load, the more it will likely increase the probability.

This observation can be used to tune the allocation process. Figure 3.16 gives an example of where *adding* a load may help get a better register allocation. If we assume that block **Q** is voracious in its use of registers, it may be that variable A has a very low chance ($\approx 0\%$) of being register resident at its load at the beginning of block **R**. However, inserting a load of A at the end of block **Q** will raise the probability for A at the beginning of **R** to 100%. With the inserted load in **Q**, the load in **R** can be removed. At the cost of inserting a load into one arm of a conditional, a load that must *always* be executed could be removed, resulting in superior code.

Our algorithm already insert loads into loop pre-headers when necessary to give a better allocation within a loop. It does this by computing probabilities as if the inserted loads were already there, and then as necessary, actually adding them. Inserting loads into less frequently executed arms of conditionals to bias allocation towards removing loads in busier parts of the code should further improve allocations.

It may be possible to isolate profitable opportunities for inserting loads by computing the probabilities that a variable will reach a load in a register along individual paths. If a low probability path traverses an infrequently executed block of code, that block may be a good location to insert a load. The infrequently executed load may greatly increase the probability of the variable reaching a subsequent, more frequently executed load. Removing the subsequent load may now be possible, and the overall effect may be positive. We believe this would be a fruitful area of future research.

3.7.2 Allocation Interactions

Probabilistic register allocation greedily allocates registers by identifying the load with the greatest estimated figure of merit and removing it. This greedy algorithm cannot, however, fully compare the "aggregate effects" of different allocation choices. For instance, it may be possible that three candidates, A, B, and C, have comparable probabilities but that an allocation for A is slightly more beneficial than for either B or C. Furthermore, assume that B and C do not interfere with each other, but they both interfere with A. In this situation, it would be possible to allocate a single register to both B and C at a greater total benefit than allocating a single register to A.

By ignoring the aggregate effects and global constraints of allocation, the greedy nature of probabilistic register allocation may make sub-optimal decisions. Because of the exponential number of ways grouping register candidates, it is not clear how to best compute aggregate benefits for purposes of guiding register allocation. The previous example suggests grouping non-conflicting register candidates as a place to start. Finding an efficient means of utilizing a measure of aggregate benefit will require future research.

3.8 Complexity

The run-time complexity of our global register allocation algorithm is $O(n^3)$ where n is the number of instructions in the procedure. This assumes that the number of loads, the number of register candidates, and the size of register-live-ranges are all proportional to the number of instructions.

Computing a single Δ-probability takes only a constant amount of work. Therefore,

the complexity of computing all the Δ-probabilities within a loop is

$$O(|loop| \times candidates).$$

After Δ-probabilities have been computed, computing the probability for any given load requires multiplying together all the Δ-probabilities in the register-live-range, which is

$$O(|register\text{-}live\text{-}range|).$$

Therefore, computing the probabilities for all the loads is

$$O(|register\text{-}live\text{-}range| \times loads).$$

Because the computation of load probabilities is done before each load removal, the algorithm takes

$$O(|register\text{-}live\text{-}range| \times loads^2).$$

Assuming the register-live-ranges are proportional in size to the program, and that the number of loads removed is too, the complexity of our algorithm is $O(n^3)$. This is a conservative estimate that reflects the complexity of the current algorithm. As noted in §3.6, it may not be necessary to recompute probabilities after each load is removed. This would decrease the complexity to $O(n^2)$.

3.9 Other Uses for Probabilities

Probabilities are a good model of the scarcity or competition for registers. The previous algorithm, adapted from Beatty's algorithm, is designed to do allocation from innermost

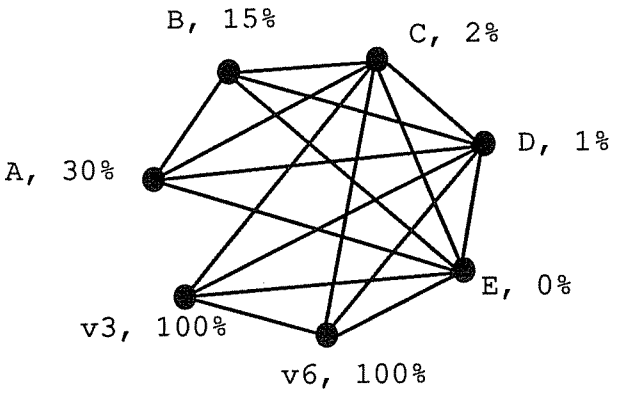


Figure 3.17: Interference Graph *with* Probabilities.

loops to outermost loops, and is tuned to make good use of probabilities. Register probabilities could be used advantageously beyond this algorithm.

3.9.1 Assisting Graph Coloring

Register probabilities are general enough to help graph-coloring algorithms site spills and split live-range. Presently graph coloring heuristics prune the interference graph by making the trivial observation that any live range with fewer conflicts than available registers can be removed from the graph and subsequently assigned a register. Probability analysis can additionally indicate, for those live ranges with too many conflicts, which will *probably* be allocated a register—thus intelligently directing further pruning of the interference graph.

For instance, the interference graph in Figure 3.2 can be updated to include probabilities. These probabilities, given in Figure 3.17 indicate the level of competition between the various register candidates—not just who is competing with whom. The probabilities were computed simply by multiplying the original Δ-probabilities for each

variable for its entire live-range.[7] The lower probabilities for D and E indicate that they are infrequently used relative to demands they place on the register allocator. The higher probabilities for A and B indicate that they put less pressure on the allocator, and would therefore be good candidates for removal. Chow's priority-based graph coloring algorithm [CH90] used the size of a live range (measured in instructions) as a crude measure of this competition. Of course, the expected benefit of allocating a register to a value must also be weighed when deciding between two candidates.

Similarly, probabilistic register allocation could be used *after* graph-coloring techniques have exploited all the trivial pruning opportunities available. If the pruning stage fails to allocate registers to every node in the graph, our probabilistic algorithm could be used to allocate registers among the (presumably) many fewer remaining register candidates.

3.9.2 Assisting Interprocedural Allocation

Interprocedural register allocation attempts to allocate registers among procedures so that procedures may pass parameters in registers, avoid saves and restores around calls, and share global values in registers.

Wall [Wal86] built a system that allocated registers interprocedurally, at link-time when the entire program was available. All local and global variables had been previously allocated to memory, and his allocator attempted to allocate these values—when most profitable—to registers. His system allocates local variables to registers so that registers will never need to be saved around calls. That is, for any possible path in the call graph, the system guarantees that at most one local variable will be assigned to

[7] The 100% probabilities for v3 and v6 assumes that local allocation has guaranteed them a register.

any one physical register.

It is no longer the case that local variables are competing for registers only against other variables (and temporaries) within the same procedure, but now they are competing against *all* potentially simultaneously live variables from *other* procedures. Wall's system chooses among different candidates based on estimates of execution frequencies of each candidate.

Register probabilities could be used interprocedurally to isolate the competition for registers among local variables of different procedures or among locals and globals. For instance, one procedure could have such high register demands that its performance would suffer greatly if registers were allocated to other procedures' locals or to globals. Such a procedure would necessarily *lower* the probabilities of any value that might be allocated a register simultaneously with it. These lower probabilities would discourage the interprocedural allocator from allocating those values to registers.

3.10 Phase-Ordering Concerns

Phase-ordering problems in a compiler exist when the order in which optimizations are applied may affect the efficacy or correctness of subsequent optimizations. In Chapter 2, we discussed the relationship between register allocation and instruction scheduling when compiling expressions for a delayed-load architecture. For that problem, we avoided the phase-ordering problems by doing register allocation in tandem with instruction scheduling.

A phase-ordering problem exists between the DLS algorithm and probabilistic register allocation. Probabilistic register allocation operates by removing loads and stores.

Computing A+B+C+D		
DLS Schedule	Remove Load of C	Rescheduled after Removing Load of C
load A, r1	**load** A, r1	**load** A, r1
load B, r2	**load** B, r2	**load** B, r2
load C, r3	{ NOP }	**load** D, r4
add r1, r2, r1	**add** r1, r2*, r1	**add** r1, r2, r1
load D, r2	**load** D, r2	**add** r1, r3, r1
add r1, r3, r1	**add** r1, r3, r1	**add** r1, r4, r1
add r1, r2, r1	**add** r1, r2, r1	

Figure 3.18: Phase-ordering problem between DLS and Probabilistic Register Allocation.

This allocation assumes that instructions have already been chosen, and that the basic blocks have been scheduled. It might seem, therefore, that DLS should logically precede the global register allocator. Unfortunately, the global register allocator, by removing loads and stores, may destroy the good schedules previously produced by DLS. Figure 3.18 demonstrates the problem encountered when DLS precedes register allocation. The left code sequence computes (A+C+C+D) in normal DLS order. The center code sequence indicates that if the "**load** C, r3" is removed, an interlock will occur before the first **add** can proceed.

If scheduling were done after probabilistic register allocation, the interlock-free schedule on the right might be found. The interlock-free schedule without the load of C requires an extra register, however, to allow the load of D to be moved earlier in the schedule. If code scheduling is done after register allocation, however, there is no guarantee that this additional register will be available.

We believe that more research is necessary to integrate instruction scheduling with *global* register allocation to avoid this phase-ordering problem.

3.11 Conclusion

Probabilities measure the competition among register candidates for scarce registers. The higher the probability a candidate will be allocated a register, the lower the scarcity faced by that candidate. By biasing allocation towards candidates with high probabilities, a register allocator uses registers sparingly—eliminating as few other competing candidates as possible from being allocated registers. Therefore, probabilistic register allocation can weigh competition along with benefit when making allocation decisions.

Using probabilities to guide global register allocation after a local allocation phase provides a simple and effective algorithm that avoids complex graph-coloring spill heuristics. It focuses attention on the problem of *allocating* registers over *assigning* registers—a weakness inherent to previous graph coloring schemes. Graph coloring algorithms awkwardly handle splitting live-ranges when registers are exhausted, but probabilistic global allocation completely subsumes this concern. Finally, because of the clean separation between local allocation and global allocation, our probabilistic algorithm allows existing excellent local register allocation and instruction scheduling algorithms to run unconstrained by global allocation policies.

Chapter 4

BURS Table Generation

Efficient *instruction selection* is difficult in compilers because CISC instruction sets often present many different choices for legally evaluating the same expression. This problem is compounded by the increasing number of new processors available—each with a unique instruction set. Because new instruction sets require new code generators, much work has been done to ease the job in retargeting a compiler. The task of creating retargetable compilers can be made more manageable if the work necessary for retargeting a compiler is isolated within the code generator (CG) and if automatic tools exist to aid in creating the retargeted CG. Such tools are known as code generator generators (CGGs).

CGGs automatically create a code generator for a particular intermediate representation (IR) from a description of the instruction set of a given processor and a description of the form of the IR. These descriptions range from being low-level patterns that, when matched against the IR, signal which machine instructions are to be emitted to very high-level descriptions of machine and IR semantics from which such patterns are deduced. A CGG allows the mechanics of finding IR–machine instruction matches to be

automated, reducing the work of creating a CG, and the likelihood of mistakes.

Possibly the simplest way to visualize and understand the complex instructions and addressing modes of a processor is to view them as expression trees in which leaves represent registers or memory locations, and internal nodes represent operations on operand values. Describing even the most complex addressing mode is simplified when such trees are used. Figure 4.1 gives an example of tree patterns.

Because of their expressive power, trees also serve as a natural IR to be generated by the front-end of a compiler. If the same domain of trees is used to describe machine instructions as is used for the IR, instruction selection for a given IR tree becomes a matter of matching instruction patterns against the generated IR such that the IR is *covered* (parsed) with adjacent patterns. Figure 4.2 shows two legal covers of the same expression tree. Many techniques are known for finding such coverings efficiently (in time proportional to the size of the IR tree). Equally important, finding a least-cost covering (based on costs associated with the patterns) is also efficient.

4.1 Overview

Tree pattern matching combined with dynamic programming can be used in code generators to create locally optimal code for expression trees [AGT89]. Code generators based on bottom-up rewrite system (BURS) theory can be extremely fast because all dynamic programming is done when the BURS automaton is built. At compile-time, it is only necessary to make two traversals of the subject tree: one bottom-up traversal to label each node with a *state* that encodes all optimal matches, and a second top-down traversal that uses these states to select and emit code. Fraser and Henry [FH91c] report that careful encodings can produce an automaton that executes fewer than 50

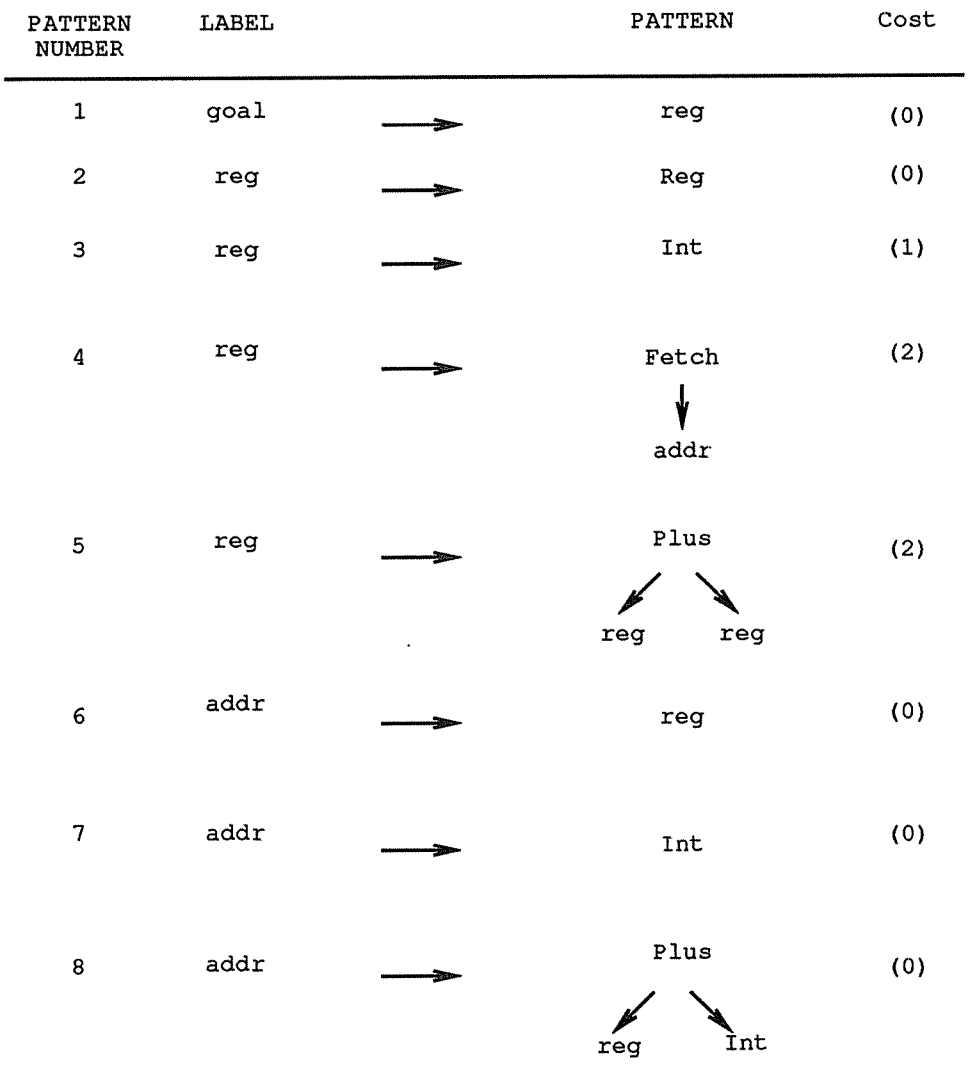


Figure 4.1: Sample Machine Instruction Templates

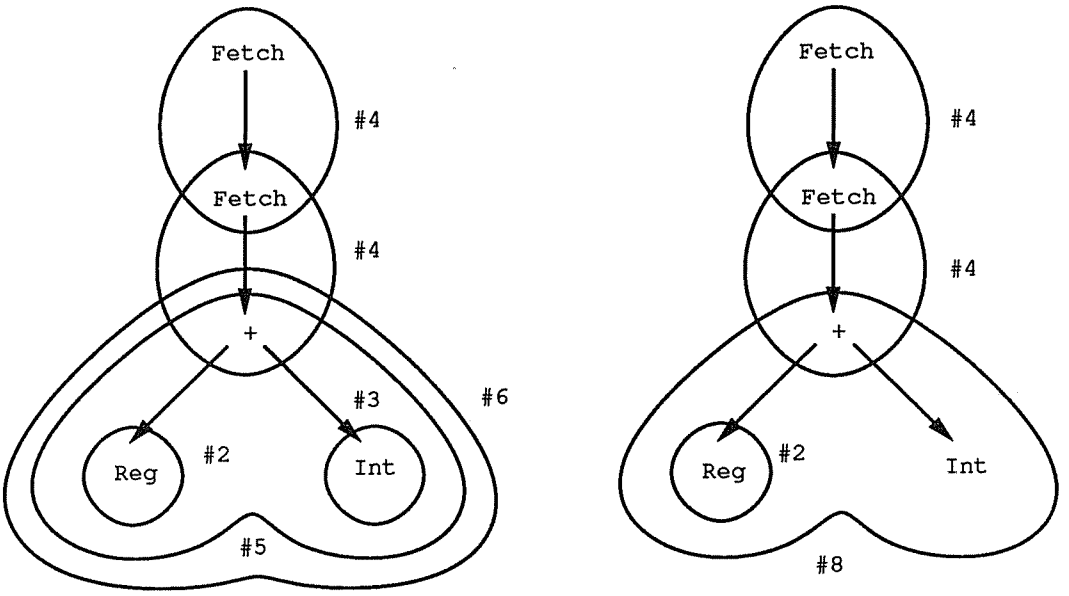


Figure 4.2: Sample Coverings of Identical Trees

VAX instructions ($\approx$ 90 RISC instructions) per node to do both traversals.

The automaton that labels the tree is a simple state transition machine. A bottom-up walk of the tree is performed and the label for any given node is determined by a table lookup given the operator at the node and the states that label each of its children. The automaton that emits code is equally simple in design. The code to be emitted is determined by the state that labels a node, and the nonterminal to which that node should be reduced—another table lookup.

Two difficulties arise in creating a BURS-style code generator: efficiently generating the states and state-transition tables (because *all* potential dynamic programming decisions are done at table-generation time, they must be done efficiently), and creating an efficient encoding of the automata for use in a compiler. A solution to the encoding problem is described by Fraser and Henry in [FH91c].

This chapter describes a new simple and efficient table generation algorithm whose implementation is an order of magnitude faster than the best current systems. Simplicity has increased, not decreased, efficiency. Efficiency has been enhanced, and tables sizes kept small, by the development of a two new techniques, *chain-rule trimming* and *triangle trimming*, for eliminating most redundant states. Triangle trimming is an uncomplicated optimization that, for complex grammars, reduces *both* the table generation time and table sizes by over 50%. We also describe optimizations that take advantage of special properties of BURS states.

4.2 Related Work

Bottom-up tree pattern matching was developed by Hoffman and O'Donnell [HO82]. Bottom-up pattern matching is the theoretically fastest possible—relying on a single bottom-up tree walk with a simple table lookup at each node in a tree to do the matching. BURS technology relies on this technology for much of its speed.

Naively generating BURS states and state-transition tables fails because the tables become too large. (The same is also true for simple Hoffman-O'Donnell bottom-up matchers *without* dynamic programming.) A typical CISC machine description will generate over 1000 states.[1] Directly encoding the transition table for a *single* binary operator would, therefore, require over 1,000,000 entries.

Fortunately, many of the rows (and columns) of bottom-up pattern-matching transition tables are identical. To exploit this redundancy, *index maps* can be used to encode much smaller tables. Index maps are vectors that map states of the automaton to *representer states* for indexing a transition table. States may share a given row or column

[1]The integer *subset* of a Motorola 68000 grammar has over 800 states (Figure 4.16).

of a transition table through a single indirection. Chase demonstrated that these maps can be produced on-the-fly during table generation so that no superfluous work need be performed [Cha87].

Pelegri-Llopart, the originator of BURS theory ([PLG88], [PL88]), encorporated Chase's ideas into a system that added cost information for dynamic programming at table generation time. In addition to recognizing that dynamic programming could be done prior to compile time, he developed the theoretical foundation for showing that the process is theoretically feasible for typical machine grammars. Pelegri's technique is not limited to finding least-cost parses of an input tree; his BURS theory also incorporated tree-rewrites. A specification could include grammar rules that allowed a matched tree to be rewritten for subsequent matches. This allowed the specification of commutativity transformations, for instance.

Subsequent BURS systems, including the techniques described here, do not allow general rewrites, but instead defer that responsibility to another phase of the compilation process. Balachandran, Dhamdhere, and Biswas [BDB90] simplified Pelegri's model by disallowing rewrite rules, and also generalized Chase's ideas to use cost information.

Henry [Hen89] developed optimization techniques to limit the number of BURS states produced during table generation. With fewer states, a smaller automaton is produced more quickly. Henry's techniques are much more aggressive than Chase's simple index map techniques, but at the cost of increased complexity. In [Hen89], Henry states, "The table builder uses space and time voraciously, even though it uses very complex algorithms designed to minimize these resources." Our algorithm generalizes and simplifies his work. Our system can be directly compared to his on a variety of machine specifications, and routinely shows a factor of 10 to 30 improvement in

	Simple Grammar				Canonical Form			
Rule#	LHS		RHS	Cost	LHS		RHS	Cost
1.	goal	→	reg	(0)	goal	→	reg	(0)
2.	reg	→	Reg	(0)	reg	→	Reg	(0)
3.	reg	→	Int	(1)	reg	→	Int	(1)
4.	reg	→	Fetch(addr)	(2)	reg	→	Fetch(addr)	(2)
5.	reg	→	Plus(reg, reg)	(2)	reg	→	Plus(reg, reg)	(2)
6.	addr	→	reg	(0)	addr	→	reg	(0)
7.	addr	→	Int	(0)	addr	→	Int	(0)
8.	addr	→	Plus(reg, Int)	(0)	addr	→	Plus(reg, n.1)	(0)
8a.					n.1	→	Int	(0)

Figure 4.3: Simple Grammar and Its Canonical Form

generation speed.

To further decrease the size of the generated tables, both Henry and Pelegri-Llopart incorporate an additional post-pass to minimize the number of states. Such a pass is essentially a DFA state minimization pass that reduces the number of states by finding states that do not differ with respect to the matches they encode or the transitions they induce. They only differ with respect to the costs that they encode—information that is not needed at compile-time.

4.3 BURS Model

The input to a BURS code generator generator is a set of rules. Each rule indicates a tree pattern, a cost, a replacement symbol, and an action. The set of all the rules is called the *grammar*. Figure 4.3 gives a small sample grammar (without actions). The replacement symbol is a *nonterminal* on the left of the rule—the linearized tree pattern it derives is on the right. In the sample, *goal, reg* and *addr* are nonterminals. In addition to nonterminals, the grammar has *operators* of varying arities. In the sample, *Reg, Int, Fetch*, and *Plus* are operators with respective arities of 0, 0, 1, and 2.

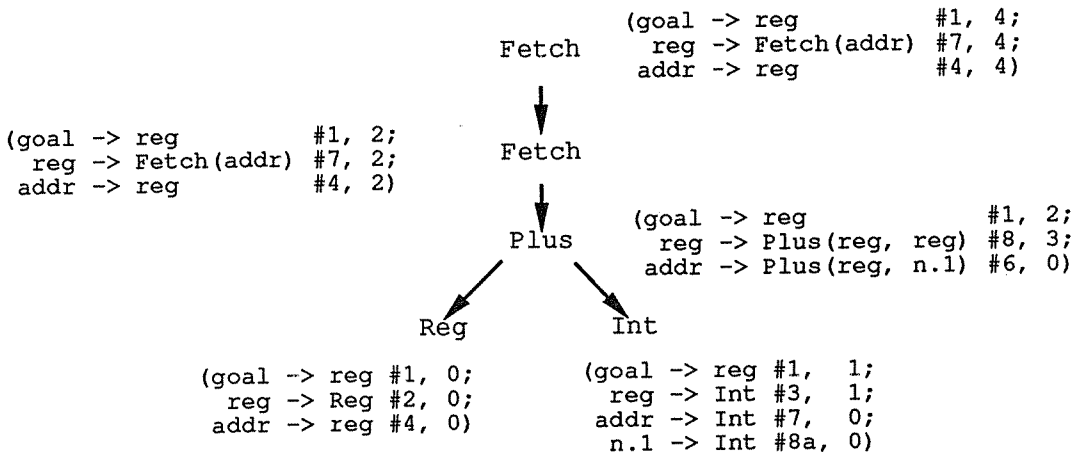


Figure 4.4: Dynamic Programming Applied to Example Tree. Each node labeled with "(Nonterminal, Rule#, Cost)"

A least-cost parse can be found using dynamic programming. By trying all matching patterns at all nodes, it is possible to remember the rules that lead to the cheapest derivation from each possible nonterminal. Figure 4.4 applies the rules in Figure 4.3 to the tree representing Fetch(Fetch(Plus(Reg, Int))). Each node is labeled with the least-cost derivation from each nonterminal.

A BURS pattern matcher finds a least-cost parse of a subject tree for the grammar that reduces to the *goal* nonterminal. Each tree node will be labeled with a state that encodes which rule is to be used when that node is to be reduced to a given nonterminal.

These states encode the information given explicitly in Figure 4.4. For example, it is possible to derive the leaf node, `Int`, from all the nonterminals. `Int` can be directly derived from the nonterminals `reg`, `addr`, and `n.1`, by directly applying the rules `#3`, `#7`, and `#8a`, respectively. The costs associated with each derivation is the cost of that particular rule. The derivation from `goal` utilizes the rule, "`goal` → `reg`," that will require that `Int` be subsequently derived from `reg`. Therefore, while the cost associated

with rule #1 is 0, the cost of the derivation is 1 — the sum of the costs of complete derivation of `Int` from `goal`.

4.3.1 Normal Form Patterns

To simplify the generation of BURS tables, all patterns are put into the *canonical form* introduced in [BDB90]. This form requires that all patterns be of the form "$n \rightarrow m$" where both n and m are nonterminals, or of the form "$n_0 \rightarrow op(n_1, \ldots, n_k)$" where n_i are all nonterminals, $k \geq 0$, and op is an operator. This canonical form does not reduce the expressiveness of the grammars—any set of rules not in canonical form can be put into canonical form by introducing new nonterminals. Putting the previous rules into canonical form gives the rules on the right of Figure 4.3.

4.4 Algorithm to Generate BURS Tables

Our method of computing the states and state transition tables is an uncomplicated *work-list* algorithm. This algorithm is outlined below in procedure *Main*(). Initially, the states corresponding to each leaf operator (arity = 0) are computed, and added to the set of known states, *States*, and to the list of states to be processed, *WorkList*. One by one, states are removed from *WorkList* and processed. For each operator with arity greater than 0, the state must be examined to determine what transitions are induced by that state when combined with each of the already processed states. These transitions may create new states to be added to the *WorkList*.

```
1   procedure Main()
2       States = ∅
3       WorkList = ∅
4       ComputeLeafStates()
5       while WorkList ≠ ∅ do
6           state = Pop(WorkList)
7           ∀ op ∈ Operators do
8               ComputeTransitions(op, state)
9           end ∀
10      end while
11  end procedure
```

4.4.1 Data Structures Used to Generate BURS Tables

The set of known states, *States*, is a table that maintains a one-to-one mapping from individual states to non-negative integers. These integers are used as indices into state transition tables via index maps.

States in a BURS code generator encode three pieces of information at any node in a subject tree: the nonterminals derived from patterns that match a rule at that node, the relative costs of those nonterminals, and which rules generated each nonterminal (at a minimal cost). Such triples are called *items*, and a collection of items describing a particular state is called an *itemset.* Itemsets are implemented as arrays of {*cost,rule*} pairs that are indexed by a nonterminal. Itemsets are, therefore, states. A cost of infinity (∞) indicates that, in this state, no rule derives the given nonterminal. The empty state ($\emptyset$) has all costs equal to infinity.

The relative costs are called *delta costs* and are always normalized so that the nonterminal with the lowest cost derivation has a delta cost of 0. Figure 4.5 gives the results of dynamic programming on the tree in Figure 4.4 after the grammar has been put into canonical form and the relative costs have been normalized. Note that the states for the two different `Fetch` nodes are identical—normalization of costs caused this to happen.

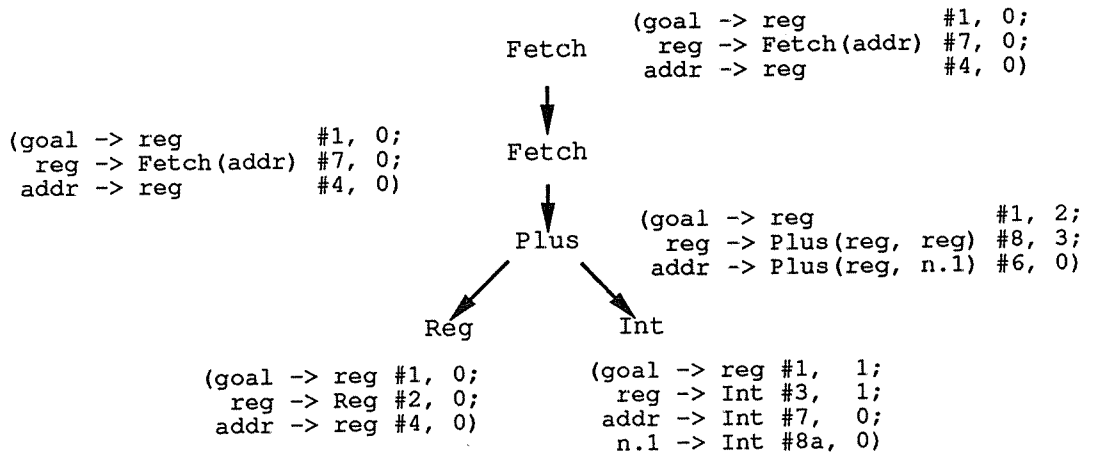


Figure 4.5: Dynamic Programming with Delta Costs. Each node labeled with "(Nonterminal, Rule#, Delta Cost)"

Without cost normalization there would be infinitely many states for this grammar.

Costs within an itemset are normalized by the routine *NormalizeCosts*() below.

```
1   procedure NormalizeCosts(state)
2       delta = min_∀i {state[i].cost}
3       ∀ n ∈ Nonterminals do
4           state[n].cost = state[n].cost − delta
5       end ∀
6   end procedure
```

4.4.2 Chain Rules

Itemsets are computed in a two-step process. *ComputeTransitions*() applies rules of the form "$n \rightarrow op(\ldots)$" to generate nonterminals in the initial itemset. Next, the algorithm computes the closure of this set by applying *chain rules*. Chain rules are rules of the form "$n \rightarrow m$" where both n and m are nonterminals. These rules may introduce new nonterminals into an itemset, or they may introduce cheaper ways of deriving nonterminals already in the set. Finding the closure of the set is done by iteratively trying all the chain rules and repeatedly applying those that add new or

cheaper nonterminals, until no changes are made. *Closure*() below implements this procedure. Because all costs are non-negative, and because a change is made only if a strictly less expensive derivation is found, this process must terminate.

One nonterminal may be derived from another by zero or more chain rule applications. The least cost derivation is denoted "$n \stackrel{*}{\Rightarrow} m$." The cost of such least cost derivations, "$Cost(n \stackrel{*}{\Rightarrow} m)$," can be computed efficiently using a shortest path algorithm.

```
1   procedure Closure(state)
2       repeat
3           ∀ r : n → m such that m ∈ Nonterminals do
4               cost = r.cost + state[m].cost
5               if cost < state[n].cost then
6                   state[n] = { cost, r }
7               end if
8           end ∀
9       until no changes to state
10  end procedure
```

4.4.3 Computing States and Transitions

The computation of the states and the state transition tables begins by generating a state for each leaf operator (with arity of 0) in the routine *ComputeLeafStates*(). These leaf states must be combined as children of each non-leaf operator, and new states will be created. Each new state is added to the *WorkList* and will be subsequently processed to determine what transitions it induces.

Computing the state to label each leaf is straightforward. Rules with a right hand side of the given leaf operator generate nonterminals directly into the itemset. Normalizing the costs and finding the closure of the itemset completes the computation of the state corresponding to the leaf operator. Figure 4.6 illustrates *ComputeLeafStates*().

```
1   procedure ComputeLeafStates()
2       ∀ leaf ∈ Leaves do
3           state = ∅                          // state[n].cost = ∞, ∀ n ∈ Nonterminals
4           ∀ r : n → leaf do
5               if r.cost < state[n].cost then
6                   state[n] = { r.cost, r }
7               end if
8           end ∀
9           NormalizeCosts(state)
10          Closure(state)
11          WorkList = Append(WorkList, state)
12          States = States ∪ {state}
13          leaf.state = state
14      end ∀
15  end procedure
```

Figure 4.6: *ComputeLeafStates*()

```
1   function Project(op, i, state)
2       pState = ∅
3       ∀ n ∈ Nonterminals do
4           if ∃ r : m → op(n_1,...,n_{i-1}, n, n_{i+1},...,n_{op.arity}) then
5               // Nonterminal n may be used in the i^th dimension of op.
6               pState[n].cost = state[n].cost
7           end if
8       end ∀
9       NormalizeCosts(pState)
10      return pState
11  end procedure
```

Figure 4.7: *Project*()

```
1   procedure ComputeTransitions(op, state)
2       ∀ i ∈ 1..op.arity do
3           pState = Project(op, i, state)
4           op.map[i][state] = pState
5           if pState ∉ op.reps[i] then
6               op.reps[i] = op.reps[i] ∪ {pState}
7               ∀ (s_1, ..., s_{i-1}, pState, s_{i+1}, ..., s_{op.arity})
                                                  such that ∀j ≠ i, s_j ∈ op.reps[j] do
8                   result = ∅
9                   ∀ r : n -> op(m_1,...,m_{op.arity}) do
10                      cost = r.cost + pState[m_i].cost + Σ_{j≠i} s_j[m_j].cost
11                      if cost < result[n].cost then
12                          result[n] = { cost, r }
13                      end if
14                  end ∀
15                  Trim(result)
16                  NormalizeCosts(result)
17                  if result ∉ States then
18                      Closure(result)
19                      WorkList = Append(WorkList, result)
20                      States = States ∪ {result}
21                  end if
22                  op.transition[s_1, ..., s_{i-1}, pState, s_{i+1}, ..., s_{op.arity}] = result
23              end ∀
24          end if
25      end ∀
26  end procedure
```

Figure 4.8: *ComputeTransitions*()

For each dimension of a non-leaf operator,[2] an index map of *representer states* is maintained. Representer states are constructed from an itemset by retaining only those nonterminals that may contribute to a match in the given dimension for the given operator ([Cha87], [BDB90]). Suppose that, for a given grammar, there is no rule with a tree pattern for the binary operator, θ, that has a left child of nonterminal n. In this case, we would project n out of any state when that state is to be examined as a possible left child (in the 1^{st} dimension) of θ.

Project() will retain only those nonterminals in a given state that may be used in determining the transitions that may be induced by that state as a given child of a particular operator. A representer state also discards the *rule* field of each item because that information does not affect transitions (only reductions). For each dimension, d, a table of representer states, $op.reps[d]$, is maintained that encodes a one-to-one mapping between those states and non-negative integers. Each dimension's $op.map[d]$ table maintains a mapping from global states to representer states ($op.map[d][s]$ is the representer state to which s maps in the d^{th} dimension of op).

Figure 4.9 illustrates the relationship between index maps and transition tables. Given the states l and r for the children of binary operator θ, an indirection is used to look up the state transition for the θ node. Figure 4.7 describes the computation of the relevant nonterminals.

Transition tables are computed based on representer states, not on the original states. This reduces transition table size because many states may map to the same representer state. At tree-matching time the cost of using this technique is the extra level of indirection necessary to compute transitions. In Figure 4.8, *ComputeTransitions*()

[2]Each operator of arity n has a transition table of n dimensions.

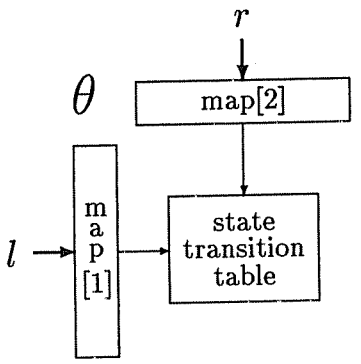


Figure 4.9: Computing Transitions for $\theta(l, r)$ Using Index Maps.

finds all the transitions that each new state induces when used in combination with other known states for a given operator.

Each representer state is checked to see if it has already been processed. If the representer state has been previously processed, then no additional work must be done. If the representer state is new, the transition table must be extended along the given dimension for all possible combinations of the representer states of other dimensions (along with this representer state). This is done by generating all such combinations and then searching for all applicable rules. Once these rules have been applied, the delta costs are normalized, and the itemset is closed. If the generated state is new, then it is added to *States* and *WorkList*.

The postponement of *Closure*() until after the check for the state's existence in *States* is an optimization justified in §4.6.4. *Trim*(), the routine responsible for reducing the number of states produced, is discussed in §4.4.4.

4.4.4 State Trimming

Many of the states created by the *ComputeTransitions*() are nearly identical. The state-generation algorithm will run faster if it can increase the likelihood that two created states will be identical. Two states can often be made identical by trimming *unessential* nonterminals from the itemset. A nonterminal is unessential (in a particular state) if it can be proven that it will never be needed to produce a least-cost cover of any subject tree. Henry devised two *ad hoc* techniques, "sibling," and "demand" trimming [Hen89], to identify when one "{ *cost, rule* }" item (representing a nonterminal) can be safely removed from a state because another item subsumes it.

Triangle Trimming

By generalizing Henry's trimming techniques, we have developed *triangle trimming* for safely removing unessential nonterminals from an itemset. Triangle trimming considers all pairs of nonterminals in a particular itemset and determines if, given their respective costs, one of the nonterminals can be removed. A nonterminal can be removed if, in all dimensions of all rules where it is applicable, the other nonterminal can be used in a different rule to generate the same resulting nonterminal at no greater cost. Informally, a nonterminal, i, can be removed from an itemset if it can be shown that everywhere i can lead to a pattern match, another nonterminal, j, in the itemset can also lead to a comparable pattern match at no greater cost.

Determining if j subsumes i requires comparisons that have a *triangular* shape (see Figure 4.10). For a given operator, θ, and in a given dimension, d, two rules must be found such that both rules represent patterns for θ, and one rule, r, can employ i as its d^{th} child, and the other rule, t, can employ j as its d^{th} child. (It is not necessary that

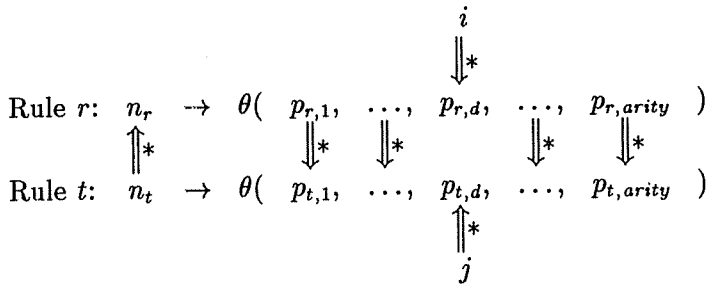


Figure 4.10: Triangle Trimming Relationship (for j to subsume i)

$$\begin{aligned} state[i].cost \;+\; r.cost \;+\; Cost(p_{r,d} \stackrel{*}{\Rightarrow} i) \;&\geq \\ state[j].cost \;+\; t.cost \;+\; Cost(p_{t,d} \stackrel{*}{\Rightarrow} j) \;&+\; Cost(n_r \stackrel{*}{\Rightarrow} n_t) \\ &+\; \textstyle\sum_{k \neq d} Cost(p_{t,k} \stackrel{*}{\Rightarrow} p_{r,k}) \end{aligned}$$

Figure 4.11: Inequality that must hold for i to be removed if j is present.

these rules use i and j directly—they may use nonterminals that are derived from i and j via chain rules.)

Since rule r reduces to nonterminal n_r, it must be shown that t can also produce n_r at no greater cost. We, therefore, start by assuming that rule r has matched. From this it can be determined if rule t can also match. Rule t can also match if its children in dimensions other than d can be derived via chain rules from the corresponding children of rule r. (All we are assuming is that r matches, therefore all we may assume in determining if $p_{t,k}$ exists for a match of rule t is whether $p_{t,k}$ may derived from $p_{r,k}$ via chain rules.)

Figure 4.10 shows how i and j, and the rules r and t must relate for j to subsume i. Once rule r is found to use i to derive n_r, a rule must be found that can employ j and can also derive n_r. Notice that for any rule r that employs i, it is only necessary to find one such rule t employing j for j to subsume i.

Subsumption is based not only on feasibility, but also on costs. A nonterminal

cannot be removed if its removal would force more expensive reductions to be found than had it been retained. For the pair of rules, r and t, in Figure 4.10, it is possible to remove i from the itemset containing j if the inequality in Figure 4.11 holds. The cost of using r is the sum of the cost of i, the cost of deriving $p_{r,d}$ from i, and the cost of r. Since our premise is only that rule r matches and that i and j are present in some itemset, the computation of the cost of using t with j to indirectly produce n_r will require not only the costs of t, j, and $p_{t,d} \stackrel{*}{\Rightarrow} j$, but will also require the costs of deriving the other $p_{t,k}$ from $p_{r,k}$ and the cost of deriving n_r from n_t.

The inequality in Figure 4.11 is the basis for finding the *minimal* cost difference between two nonterminals to allow one of them to be removed for a given rule. In general, to safely remove i, it is necessary to examine *all* contexts in which i can be used and find the cost difference that is sufficient to guarantee that i can be removed based on the relative costs of i and j. In Figure 4.12, *Triangle*(), calculates this minimal difference for any pair of nonterminals. (When it is impossible for nonterminal j to be used in place of i, regardless of their respective costs, *Triangle*() returns ∞.)

Chain Rule Trimming

Two states are identical if they represent the same nonterminals at the same costs with each respective nonterminal generated by the same rule. Triangle trimming removes nonterminals from states whenever possible, thereby eliminating the possibility that two states differ on the particular costs or rules involving those nonterminals. To further minimize the number of states, it is necessary to bias the algorithm towards using the same rules whenever possible. Biasing the algorithm towards using chain rules whenever possible increases the likelihood that two states will have used the same rules to derive

```
// Compute C, such that if state[i].cost ≥ state[j].cost + C
// then i can safely be removed from state.
1   function Triangle(i, j)
2       if i = Goal then
3           return ∞        // Do not remove the goal nonterminal.
4       end if
5       Max = −∞
6       ∀ n ∈ Nonterminals−{i} do
7           if Max < Cost(n ⇒* j) − Cost(n ⇒* i) then
8               Max = Cost(n ⇒* j) − Cost(n ⇒* i)
9           end if
10      end ∀
11      ∀ op ∈ Operators do
12          ∀ d ∈ 1..op.arity do
13              ∀ r : n_r → op(p_{r,1},...,p_{r,op.arity}) do
14                  C_i = Cost(p_{r,d} ⇒* i)
15                  if C_i < ∞ then
16                      LocalMin = ∞
17                      ∀ t : n_t → op(p_{t,1},...,p_{t,op.arity}) do
18                          C_{r,t} = Cost(n_r ⇒* n_t)
19                          C_j = Cost(p_{t,d} ⇒* j)
20                          C_k = Σ_{k≠d} Cost(p_{t,k} ⇒* p_{r,k})
21                          C = C_{r,t} + C_j + C_k + t.cost − r.cost − C_i
22                          if C < LocalMin then
23                              LocalMin = C
24                          end if
25                      end ∀
26                      if LocalMin > Max then
27                          Max = LocalMin
28                      end if
29                  end if
30              end ∀
31          end ∀
32      end ∀
33      return Max
34  end procedure
```

Figure 4.12: *Triangle*()

```
1   procedure Trim(state)
2       ∀ n ∈ state do
3           ∀ m ∈ state (m≠ n) do
4               C = Cost(n ⇒* m)
5               if state[n].cost ≥ state[m].cost + C then
6                   state[n] = { ∞, ⊥ } // Remove n from state.
7               end if
8           end ∀
9       end ∀
10      ∀ n ∈ state do
11          ∀ m ∈ state (m≠ n) do
12              C = Triangle(n, m)
13              if state[n].cost ≥ state[m].cost + C then
14                  state[n] = { ∞, ⊥ } // Remove n from state.
15              end if
16          end ∀
17      end ∀
18  end procedure
```

Figure 4.13: *Trim*()

a given nonterminal. This bias can be forced by removing nonterminal entries from an itemset prior to closure when it can be determined that *Closure*() will restore those nonterminals at an equal or lesser cost using chain rules.

In Figure 4.13, *Trim*(), uses both triangle and chain rule trimming to prune nonterminals from itemsets so that they will be more likely to be identical, thereby reducing the size of the generated tables and the table generation time.

Fully General Trimming

Nonterminal trimming does not need to be constrained to looking at pairs of nonterminals as it was in triangle and chain rule trimming. While it may not be the case that a single nonterminal in an itemset subsumes any other, it may be the case that some *set*

of nonterminals subsumes another.

One can ask the simple question, "Given *all* the nonterminals, n_i, in state N, can we safely remove nonterminal n_k?" This could be answered by attempting, one by one, to remove nonterminals from the itemset and determining by analysis similar to triangle trimming if that removal would force more costly matches to be found. If not, the nonterminal can be safely removed.

We do not know how much, if any, general trimming would reduce the number of states. The general approach to state trimming was not attempted because it is significantly more expensive than triangle trimming. Because triangle trimming tests pairs of nonterminals, the relative costs necessary for subsumption can be cached for reuse easily (see §4.6.3). There is no simple relationship based on a set of nonterminals that can be so easily stored and accessed.

4.5 Diverging Grammars

Because all dynamic programming is done at compile-compile time, it is necessary to anticipate *all* possible trees, and generate states that can label the nodes of those trees. To do this, there must be only a finite number of states. Grammars that do not produce a finite number of states are said to *diverge* [PL88].

A grammar diverges when it is possible for the derivation costs of a pair of nonterminals in the same state to become arbitrarily distant. To prevent the BURS table generation algorithm from attempting to enumerate an infinite set of states for diverging grammars, a simple threshold test is used. A test is inserted into the normalization procedure (*NormalizeCosts*()) to determine the greatest cost differential between nonterminals in any given state. If that differential is above the threshold value, the grammar

is rejected as "probably diverging."

Fortunately, code generation grammars do not typically diverge. This is because the nonterminals usually describe data values (*e.g.*, registers, data, addressing modes) that can be interchanged at a bounded cost. For instance, it is unlikely that the cost of computing a value into a register could be arbitrarily more expensive than computing a value into memory since there almost certainly is a store instruction of fixed cost.

4.6 Speed Optimizing Techniques

The previous routines provide many opportunities for speed optimization. Some of the improvements are general techniques not specific to BURS table generation; other improvements rely on subtle knowledge of BURS table generation.

4.6.1 Attempt Cheaper Alternatives First

It may appear that the two sets of nested loops in *Trim*() could be jammed into a single pair of nested loops for improved efficiency. Both loops have the intended side-effect of removing nonterminals from the states. Since the loops iterate over only the nonterminals that remain in the state, the second set of loops will normally iterate fewer times than the first set. Because triangle trimming is an expensive operation relative to chain rule trimming, it is more efficient to remove all possible nonterminals via chain rule trimming and then attempt triangle trimming only on the remaining nonterminals.

4.6.2 Precompute Values

In the previous routines, many situations exist where values can be computed once and used many times. For instance, *Project*() requires the knowledge of which nonterminals

can appear in the i^{th} dimension of operator *op*. Because this list is invariant for a given rule set, it can be computed once and used repeatedly.

Efficiency is also enhanced if the list of rules is partitioned by the operator of the pattern, so that *ComputeTransitions*() will only iterate over the list of applicable rules.

The cost of transitive closure rules ($Cost(n \stackrel{*}{\Rightarrow} m)$) is precomputed advantageously since it is used often by *Trim*() and *Triangle*().

4.6.3 Lazy Computations

There are $O(N^2)$ possible pairs of nonterminals that may be used in a call to *Triangle*(), but in practice only very few pairs are ever used. Our original implementation precomputed the results of calling *Triangle*() with all possible combinations of nonterminals and then used table lookup for these values. Using this strategy, *Triangle*() consumed over 75% of the execution time generating tables for a VAX grammar. With 179 nonterminals in the (canonical form) grammar, *Triangle*() was called 32041 times, but fewer than 1000 of those values were ever referenced! Changing the program to compute those values by need increased the speed tremendously. Once computed, these values are cached for subsequent calls with the same arguments.

4.6.4 Defer Closure

If two itemsets are equal before closure, then they must be equal after closure. Because two itemsets are chain-rule trimmed before closure, it is also the case that if two itemsets are equal after closure, they must have been equal before closure. By maintaining both pre-closure and post-closure copies of an itemset in a table, we can check for the existence of an itemset in the table by comparing their pre-closure representations. This allows

the closure computation to be deferred until it is known that the state is indeed new and must be added to the table.

4.6.5 Itemset Equivalence

Determining whether an itemset is already in a table of states is an expensive operation, and this test is done for every entry in every transition table. The integer subset 68000 grammar required over 425,000 calls to determine itemset equivalence. Making itemset equivalence testing efficient is extremely important. For two itemsets to be equal, they must be equal for all of their items. Fortunately, two observations make testing for equivalence much more efficient: two itemsets created as members of transition tables for different operators can never be equal, and for any given operator it is only necessary to compare the entries corresponding to the left-hand sides of the rules for that operator.

By keeping a reference to the generating operator as part of an itemset's representation, many itemsets can be determined to be unequal by recognizing that those entries differ. Should those entries be the same, it is only necessary to check that the nonterminal entries for the relevant nonterminals are equal for both itemsets. This check must be done after the states have been trimmed.

The same routines are used to implement the global *States* table, and each of the local *op.reps*[] tables. These tables are implemented as hash tables. Computing the hash function is also made more efficient by examining only the relevant nonterminals.

Calling *NormalizeCosts*() after *Trim*(), but before *Closure*(), allows it to limit the nonterminals it must inspect. Again, the same nonterminals that are relevant to determining itemset equivalence are those that must be normalized prior to a call to *Closure*().

4.6.6 Specialize Memory Allocation

Our program allocates and deallocates an enormous amount of memory during the computation of the itemsets and transition tables. The primary source of allocation and deallocation of memory in the algorithm is the tentative allocation of itemsets by *ComputeTransitions*() and *Project*(). Only after the itemset is allocated and computed can it be determined if an equivalent state has already been seen, thereby allowing the deallocation of the itemset. Redundant itemsets really *must* be deallocated—for a 68000 grammar the program computed over 100,000 redundant itemsets.

Fortunately, knowledge of the the allocation/deallocation pattern of particular data can lead to very efficient memory management [Han90]. This is the case with itemsets. Itemsets, after allocation, are computed and then either retained forever or immediately released. It can never be the case, therefore, that two itemset deallocations occur sequentially without an intervening allocation. This allows the creation of specialized deallocation and allocation routines for itemsets. The deallocation routine simply maintains a reference to the last discarded itemset, and does not return the space to the heap. Allocation checks this reference, and if the reference is not null, it returns the reference to the previously deallocated value (and clears the reference); only if the reference is null does the allocator request space from the heap.

4.6.7 Minimize space

On a machine without enormous amounts of RAM, it is important to avoid over-allocating memory and thrashing. The single biggest user of memory is the itemset representation for all of the computed states. Itemsets are kept as small as possible by minimizing the number of nonterminals in the canonical form grammar. A naive

translation of a grammar into canonical form may produce too many nonterminals if it creates different nonterminals that represent identical patterns. It is important (and easy) to reuse previously created nonterminals.

4.7 Unprofitable Optimizations

Three additional techniques were not implemented because either the speed-up did not merit the additional complexity, or because the resulting code would only reduce the number of states, without also speeding up the code.

4.7.1 Closure Speedup

Because least-cost transitive chain rules are precomputed for use by *Trim*(), they are available for speeding up the *Closure*() routine. *Closure*(), however, represents less than 4% of the execution time of the program, and using these transitive rules only speeds that routine by 10-20%.

4.7.2 Post-pass State Minimization

It is possible to further eliminate states after they and the transition tables have been generated by isolating and removing states that differ only in the respective costs of each constituent nonterminal. State minimization for BURS is similar to DFA state minimization. Because state minimization is a post-pass, it cannot make the program faster—it must make it slower.[3] We decided the space savings was not worth the additional complexity or time and, therefore, did not attempt to add a state minimization

[3]Henry [Hen89] found that the additional time for the post-pass was negligible (< 1%) in his system.

pass.

4.7.3 Normalize Specialization

When an itemset is normalized, the relative costs of all the nonterminals are retained. This is unduly conservative because certain nonterminals can never be used in the same context, and could, therefore, be independently normalized within the same itemset. It is possible to partition the set of nonterminals based on whether they can be used in the same context (*i.e.* in the i^{th} dimension of a given operator, or as part of the same chain rule). Once the nonterminals are partitioned, each partition can be independently normalized.

The hope was that this would cause more identical states to be found because differences between elements of different partitions would now be irrelevant. Unfortunately, only the VAX grammar showed any reduction in the number of states—2 states were eliminated from over 500.

4.8 Output

The table generator must output two sets of data: the state transition tables for labeling the subject tree, and a mapping from (states×nonterminals) to rules for reducing the matched tree and emitting code.

For the transition tables, it is necessary to output both the n-dimensional transition tables (*op.transition*) and the mappings from states to representer states for each dimension (*op.reps*[*d*]) since the transition tables are indexed by representer states. For leaf nodes, it is only necessary to give the mapping from the node to its unique state (*leaf.state*).

Function	Lines (C/Yacc)
Table Generation	1981
Front End	633
Table Output	1345
Total	3959

Figure 4.14: Code size for our BURS table generator

The reduction mapping is a table of all the states (*States*) and the *rule* fields that correspond to each nonterminal. These fields indicate which rule produces the given nonterminal. There is no need for the *cost* field at compile-time.

4.9 Implementation Results

Our algorithm has been implemented in a system called "BURG" [FHP91]. The input has two parts: a description of the operators (including the arity and identifying value of each), and a list of grammar rules. The operators are limited to being nullary (leaf), unary, or binary. (The arity was limited because the intended application required only nullary, unary, and binary operators.) Each rule includes an arbitrarily complex pattern, the nonterminal the pattern derives, its cost, and a unique external rule number (for identification). The front end of the table generator puts the rules into canonical form.

As output the program creates C routines and tables for labeling and reducing a subject tree. The program can output either a simple table-driven tree-labeler and reducer, or a hard-coded labeler and reducer. The hard-coded routines incorporate the time and space saving techniques in [FH91c].

The entire program is under 4000 lines of code that splits evenly between table generation routines and input/output routines. Figure 4.14 gives the number of lines of code used to implement the table generator.

Grammar		Time (sec.)		Ratio
Machine	Rules	Henry's	Ours	
vax	291	467.7	14.4	32
MIPS	136	21.4	0.6	36
vax.bwl	524	146.8	15.5	9
mot.bwl	462	251.5	14.4	14

Figure 4.15: Timings

Grammar		States		Ratio
Machine	Rules	Henry's	Ours	
vax	291	1017	1015	1.00
MIPS	136	125	125	1.00
vax.bwl	524	493	586	0.84
mot.bwl	462	499	838	0.60

Figure 4.16: Number of States

Our program runs quickly on both simple and complex inputs. We compare our system to Henry's table generator that was derived from the CodeGen system [Hen89]. His system consists of over 20,000 lines of C code. It is not clear, however, how much of this code is a direct consequence of algorithm design, and how much is an indirect consequence of the fact that his BURS system was derived from the much bigger CodeGen distribution.

Figure 4.16 gives a description of 4 sample input grammars and the execution times for each system on each grammar. The first two grammars (used to generate code generators for `lcc` [FH91a]) are for the VAX and the MIPS R3000 RISC processor. Two others that were developed as part of the CodeGen project are integer (byte, word, and long) subsets of the VAX and Motorola 68000 processors. The timings were taken on a DECstation 5000 with 96Mb of RAM.[4]

[4]The timings are *more* favorable towards our system on machines with limited amounts of RAM.

Rule #	LHS		Pattern	Cost
1.	integer	→	ADD(integer, integer)	(0)
2.	real	→	ADD(real, real)	(0)
3.	set	→	ADD(set, set)	(0)
4.	real	→	integer	(1)

Figure 4.17: Inference Rules for Pascal's "+"

The differences in the number of generated states between the two systems for the CodeGen grammars can be attributed to the presence of a state minimization post-pass in Henry's system that is not present in our system.

4.10 Other Applications of BURS

BURS technology has applications outside of instruction selection. For instance, BURS can be used to do simple type inferencing, data structure auditing, and tree simplification.

4.10.1 Simple Type Inferencing

Waite has used BURG to automate simple type inferencing [Wai91]. In many Algol-like languages, arithmetic operators are overloaded and may operate on different types. For example, in Pascal, "+" may operate on sets, reals, or integers—but both operands must be the same type. To add a real and an integer, the compiler must realize that the integer must be converted to a real before the addition. However, the compiler must not convert two integer operands to reals.

BURS simplifies the process of determining which instance of the overloaded operator must be used, and what conversions (if any) must be performed. The BURS grammar for this simple inference system is given in Figure 4.17. Given this grammar,

Rule #	LHS		Pattern	Cost
1.	int	→	ADDI(int, int)	(0)
2.	double	→	ADDD(double, double)	(0)
3.	double	→	CVID(int)	(0)

Figure 4.18: Type Rules for `lcc`'s Intermediate Representation

a BURS tree matcher will try to find a least-cost match of the tree. The patterns chosen will indicate which type of "addition" must be used. If the rule "real → integer" is used, then an integer must be converted to a real at that location. Because the conversion rule has a cost greater than zero, it will not be used unless necessary to find a legal parse of the tree.

4.10.2 Data Structure Auditing

BURS pattern matchers can also be used to determine if a tree has *any* legal parses. If the underlying grammar defines all *legal* tree structures, this can be used to quickly audit trees to ensure that they are not malformed.

In an experiment, a BURS pattern matcher audited the intermediate representation generated by the front-end of `lcc` [FH91b], an ANSI C compiler. The IR trees were tested to see if they were correctly formed with respect to basic types. A small portion of the grammar is given in Figure 4.18. If the matcher finds a parse, then the expression tree is legal, otherwise it is malformed.

After running the experiment on about 10,000 lines of C, a few trees were found that did not parse. The problem was not in `lcc`, it was in the IR documentation. The documentation did not fully describe all the legal combinations of intermediate operators and types. The experiment was intended to search for bugs in the implementation, and succeeded in finding omissions in the documentation.

Rule #	LHS		Pattern	Cost
1.	int	→	ADDI(int, int)	(1)
2.	int	→	NEGI(int)	(1)
3.	int	→	ADDI(int, NEGI(int))	(0)
4.	int	→	ADDI(NEGI(int), int)	(0)

Figure 4.19: Simplification Rules for `lcc`'s Intermediate Representation

4.10.3 Tree Simplification

Tree pattern matching can also be used to find opportunities for tree simplification in a compiler. Patterns can be used to represent opportunities for tree modifications that will result in simpler or more efficient code.

A simple modification would be to substitute subtraction for the addition of a negated integer. The example in Figure 4.19 gives rules for parsing the ADDI and NEGI operators. When a BURS pattern matcher finds a parse for an expression tree, it will choose to use rules #3 and #4 whenever possible since they have a lower cost than the composition of rules #1 and #2. A subsequent simplification pass would isolate these rules and perform the necessary tree modifications. Since the matchers are automatically generated, it is a simple matter to incrementally build the patterns that lead to simplifications.

4.11 Related Systems

Other code generation systems based on tree pattern matching and dynamic programming have been developed. They differ primarily in what technology they use to do tree pattern matching, and in the fact that they do dynamic programming at compile time rather than compile-compile time.

4.11.1 Twig

Aho, Ganapathi, and Tjiang [AGT89] created a tree manipulation language and system called *Twig*. Given a specification of tree patterns and associated costs, Twig generates a tree automaton that will find the least-cost cover of a subject tree. Twig uses fast top-down Hoffmann-O'Donnell [HO82] pattern matching in parallel with dynamic programming to find the least-cost cover in $O(patno \times |tree|)$ time (where *patno* is the number of patterns in the grammar, and $|tree|$ is the size of the tree to be parsed).

The costs associated with patterns in Twig are more general than those afforded by (any) BURS system. Twig may compute the cost of a pattern dynamically—depending on semantic information available at compile-time. This flexibility further allows Twig to *abort* certain matches if semantic predicates are not satisfied. Thus, the applicability of Twig's patterns is *context-sensitive*. BURS cannot have this flexibility since all costs must be compile-compile time constants to precompute dynamic programming decisions.

4.11.2 BEG

A code generator generator based on tree pattern matching was developed by Emmelmann, *et al.* [ESL89]. The Back End Generator (BEG) uses naive pattern matching to find pattern matches within the tree IR to do instruction selection. The least-cost cover of the tree is found using dynamic programming techniques that are essentially identical to Twig's. Like Twig, BEG can guard patterns with semantic predicates.

A BEG specification, in addition to having instruction patterns, includes a description of the register set of the target machine. This specification is used to generate the register allocator. Two different types of register allocators may be generated: a simple on-the-fly allocator, and a more complex post-pass allocator that processes the cover

tree prior to emitting instructions. They found the code quality and code generation times to be comparable to their hand-written CGs.

Both Twig and BEG have the advantage over BURS of being able to incorporate semantic information into pattern matching and dynamic programming. However, they generate pattern matchers that are significantly slower than pattern matchers based on BURS technology. This is because they use slower pattern matching technology (either top-down or naive) *and* they do dynamic programming at compile time. For a VAX grammar, experiments indicate that BURS code generators are about 7 times faster than highly-tuned compile-time dynamic programming.[5] Switching from BURS to the slower system decreases `lcc`'s speed by 15%.

4.12 Conclusion

The BURS table generation algorithm presented is a simple and efficient method of producing BURS tables. To the best of our knowledge our system is significantly faster than any other BURS system. The prototype implementation required fewer than 2000 lines of C code for producing the BURS automata. It was able produce these tables over 30 times more quickly than the previous "state of the art" optimizing system. Our system does not sacrifice table compaction optimizations to achieve this speed—to the contrary, the compaction techniques increase the overall speed of the implementation by reducing the number of states that must be examined.

The algorithm employs only simple data structures and routines to generate these tables quickly. We believe that, to a large degree, this design simplicity increases efficiency. To further increase speed, optimizations that exploit the specific nature of

[5]The system used for comparison generates faster code generators than Twig.

BURS table generation were isolated and are described here.

To reduce the number of states created a new technique of trimming states, triangle trimming, has been developed to isolate nonterminals that can be removed from a state. This trimming provides a many-fold reduction in the number of states and a commensurate speed-up in table generation.

Chapter 5

Conclusions and Future Work

This thesis makes the following advances in code generation theory and technology: (a) development of an optimal instruction scheduler and register allocator for delayed-load architectures; (b) invention and implementation of *probabilistic register allocation*, a global register allocation heuristic that we believe to be simpler and more effective than widely-used graph-coloring techniques; (c) design and implementation of new and highly optimized techniques for producing retargetable BURS instruction selectors. Or, more simply stated, this thesis attacks "the three main problems in code generation": instruction scheduling, register allocation, and instruction selection.

The Delayed-Load Scheduling (DLS) algorithm presented in Chapter 2 is an optimal, linear-time solution to the problem of scheduling instructions *and* allocating registers for evaluating expression trees on architectures with load delays of one cycle. The algorithm minimizes both execution time and register use and runs in time proportional to the size of the expression tree. For less restricted problems, optimal register allocation and instruction scheduling are NP-complete problems, and must be handled by slow, and often complicated, heuristics. DLS, however, represents a simple and fast solution for a

realistic class of modern RISC architectures. In addition, the algorithm is simple; it fits on one page. When the delay is greater than a single cycle, DLS works as an excellent heuristic while maintaining the same simplicity.

We believe that the integrated instruction scheduling and register allocation model developed for DLS provides a useful foundation extending the DLS algorithm to handle arbitrary pipeline delays. A more powerful algorithm will be necessary to schedule operations with long latencies such as floating point multiplications and divisions. Covering long latencies (tens of cycles) will certainly require basic block, and possibly inter-block, level scheduling. Integrating DLS register allocation into an inter-block scheduler will be necessary to effectively allocate registers and schedule instructions at the procedure level.

To help automate instruction selection, this thesis describes a simple and efficient mechanism for generating Bottom-Up Rewrite System (BURS) automata to be employed in code generators. BURS technology produces extremely fast code generators by doing all dynamic programming before code generation. Thus, the generation process can be slow. To make BURS technology more attractive much effort has gone into reducing the time to produce BURS code generators. These complex techniques often require a long time to process a complex machine description (over 10 minutes on a fast workstation). The algorithms in Chapter 4 improve the efficiency of BURS table generation algorithms so that BURS technology is now a more attractive method for instruction selection. The new optimized techniques have increased the speed to generate BURS code generators by a factor of 10 to 30. In addition, the algorithms simplify previous techniques, and were implemented in fewer than 2000 lines of C.

While tree pattern matching is a excellent model for instruction selection, it has

limitations. To model sharing, intermediate representations are more naturally DAG-structured; to express side-effects, machine instructions are also more naturally DAG-structured. We believe that extending BURS pattern matching technology in the future to DAGs could create more expressive and powerful code generator-generators for more realistic machine models.

The dominant paradigm in modern global register allocation is graph coloring. Unlike graph-coloring, our technique, *probabilistic register allocation* (described in Chapter 3), is unique in its ability to quantify the likelihood that a particular value might be allocated a register *before* allocation actually completes. By computing the likelihood that a value will be assigned a register by a register allocator, register candidates that are competing heavily for scarce registers can be isolated from those that have less competition. These probabilities allow the register allocator to concentrate its efforts where benefit is high *and* the likelihood of a successful allocation is also high. Probabilistic register allocation also avoids backtracking and complicated live-range splitting heuristics that plague graph-coloring algorithms.

Probabilistic register allocation can be extended to assist interprocedural register allocation. By computing probabilities for register candidates that compete for registers *between* procedures, better interprocedural register allocations should be isolated. Better interprocedural allocations will avoid much of the expense of program overhead by passing parameters in registers, avoiding call/save sequences around calls, and allocating global variables to registers.

Appendix A

BURG Reference Manual

Originally published as

Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — fast optimal instruction selection and tree parsing. SIGPLAN Notices, *27(4), April 1991.*

A.1 Overview

BURG is a program that generates a fast tree parser using BURS (Bottom-Up Rewrite System) technology. It accepts a cost-augmented tree grammar and emits a C program that discovers in linear time an optimal parse of trees in the language described by the grammar. BURG has been used to construct fast optimal instruction selectors for use in code generation. BURG addresses many of the problems addressed by TWIG [AG85, App87], but it is somewhat less flexible and much faster. BURG is available via anonymous `ftp` from `kaese.cs.wisc.edu`. The compressed `shar` file `pub/burg.shar.Z` holds the complete distribution.

This document describes only that fraction of the BURS model that is required to

```
%{
#define NODEPTR_TYPE treepointer
#define OP_LABEL(p) ((p)->op)
#define LEFT_CHILD(p) ((p)->left)
#define RIGHT_CHILD(p) ((p)->right)
#define STATE_LABEL(p) ((p)->state_label)
#define PANIC printf
%}
%start reg
%term Assign=1 Constant=2 Fetch=3 Four=4 Mul=5 Plus=6
%%
con:  Constant                = 1 (0);
con:  Four                    = 2 (0);
addr: con                     = 3 (0);
addr: Plus(con,reg)           = 4 (0);
addr: Plus(con,Mul(Four,reg)) = 5 (0);
reg:  Fetch(addr)             = 6 (1);
reg:  Assign(addr,reg)        = 7 (1);
```

Figure A.1: A Sample Tree Grammar

use BURG. Readers interested in more detail might start with [BDB90]. Other relevant documents include [Kro75, HO82, HC86, Cha87, PLG88, PL88, BMW87, Hen89, FH91c, Pro92].

A.2 Input

BURG accepts a tree grammar and emits a BURS tree parser. Figure A.1 shows a sample grammar that implements a very simple instruction selector. BURG grammars are structurally similar to YACC's. Comments follow C conventions. Text between "%{" and "%}" is called the *configuration section*; there may be several such segments. All are concatenated and copied verbatim into the head of the generated parser, which is called BURM. Text after the second "%%", if any, is also copied verbatim into BURM, at

the end.

The configuration section configures BURM for the trees being parsed and the client's environment. This section must define `NODEPTR_TYPE` to be a visible typedef symbol for a pointer to a node in the subject tree. BURM invokes `OP_LABEL(p)`, `LEFT_CHILD(p)`, and `RIGHT_CHILD(p)` to read the operator and children from the node pointed to by `p`. It invokes `PANIC` when it detects an error. If the configuration section defines these operations as macros, they are implemented in-line; otherwise, they must be implemented as functions. The section on diagnostics elaborates on `PANIC`.

BURM computes and stores a single integral *state* in each node of the subject tree. The configuration section must define a macro `STATE_LABEL(p)` to access the state field of the node pointed to by `p`. A macro is required because BURG uses it as an lvalue. A C `short` is usually the right choice; typical code generation grammars require 100–1000 distinct state labels.

The tree grammar follows the configuration section. Figure A.2 gives an EBNF grammar for BURG tree grammars. Comments, the text between "`%{`" and "`%}`", and the text after the optional second "`%%`" are treated lexically, so the figure omits them. In the EBNF grammar, quoted text must appear literally, `Nonterminal` and `Integer` are self-explanatory, and `Term` denotes an identifier previously declared as a terminal. $\{X\}$ denotes zero or more instances of X.

Text before the first "`%%`" declares the start symbol and the terminals or operators in subject trees. All terminals must be declared; each line of such declarations begins with `%term`. Each terminal has fixed arity, which BURG infers from the rules using that terminal. BURG restricts terminals to have at most two children. Each terminal is declared with a positive, unique, integral *external symbol number* after a "`=`". `OP_LABEL(p)`

```
grammar:  {dcl} '%%' {rule}

dcl:      '%start' Nonterminal
dcl:      '%term' { Identifier '=' Integer }

rule:     Nonterminal ':' tree '=' Integer cost ';'
cost:     /* empty */
cost:     '(' Integer { ',' Integer } ')'

tree:     Term '(' tree ',' tree ')'
tree:     Term '(' tree ')'
tree:     Term
tree:     Nonterminal
```

Figure A.2: EBNF Grammar for Tree Grammars for BURG

must return the valid external symbol number for p. Ideally, external symbol numbers form a dense enumeration. Non-terminals are not declared, but the start symbol may be declared with a line that begins with %start.

Text after the first "%%" declares the rules. A tree grammar is like a context-free grammar: it has rules, non-terminals, terminals, and a special start non-terminal. The right-hand side of a rule, called the *pattern*, is a tree. Tree patterns appear in prefix parenthesized form. Every non-terminal denotes a tree. A chain rule is a rule whose pattern is another non-terminal. If no start symbol is declared, BURG uses the non-terminal defined by the first rule. BURG needs a single start symbol; grammars for which it is natural to use multiple start symbols must be augmented with an artificial start symbol that derives, with zero cost, the grammar's natural start symbols. BURM will automatically select one that costs least for any given tree.

BURG accepts no embedded semantic actions like YACC's, because no one format suits all intended applications. Instead, each rule has a positive, unique, integral *external*

rule number, after the pattern and preceded by a "=". Ideally, external rule numbers form a dense enumeration. BURM uses these numbers to report the matching rule to a user-supplied routine, which must implement any desired semantic action; see below. Humans may select these integers by hand, but BURG is intended as a *server* for building BURS tree parsers. Thus some BURG clients will consume a richer description and translate it into BURG's simpler input.

Rules end with a vector of non-negative, integer costs, in parentheses and separated by commas. If the cost vector is omitted, then all elements are assumed to be zero. BURG retains only the first four elements of the list. The cost of a derivation is the sum of the costs for all rules applied in the derivation. Arithmetic on cost vectors treats each member of the vector independently. The tree parser finds the cheapest parse of the subject tree. It breaks ties arbitrarily. By default, BURG uses only the *principal cost* of each cost vector, which defaults to the first element, but options described below provide alternatives.

A.3 Output

BURM traverses the subject tree twice. The first pass or *labeller* runs bottom-up and left-to-right, visiting each node exactly once. Each node is labeled with a state, a single integer that encodes all full and partial optimal pattern matches viable at that node. The second pass or *reducer* traverses the subject tree top-down. The reducer accepts a tree node's state label and a *goal* non-terminal — initially the root's state label and the start symbol — which combine to determine the rule to be applied at that node. By construction, the rule has the given goal non-terminal as its left-hand side. The rule's pattern identifies the subject subtrees and goal non-terminals for all recursive visits.

Here, a "subtree" is not necessarily an immediate child of the current node.[1] Patterns with interior operators cause the reducer to skip the corresponding subject nodes, so the reducer may proceed directly to grandchildren, great-grandchildren, and so on. On the other hand, chain rules cause the reducer to revisit the current subject node, with a new goal non-terminal, so x is also regarded as a subtree of x.

As the reducer visits (and possibly revisits) each node, user-supplied code implements semantic action side effects and controls the order in which subtrees are visited. The labeller is self-contained, but the reducer combines code from BURG with code from the user, so BURM does not stand alone.

The BURM that is generated by BURG provides primitives for labelling and reducing trees. These mechanisms are a compromise between expressibility, abstraction, simplicity, flexibility and efficiency. Clients may combine primitives into labellers and reducers that can traverse trees in arbitrary ways, and they may call semantic routines when and how they wish during traversal. Also, BURG generates a few higher level routines that implement common combinations of primitives, and it generates mechanisms that help debug the tree parse.

BURG generates the labeller as a function named `burm_label` with the signature

```
extern int burm_label(NODEPTR_TYPE p);
```

It labels the entire subject tree pointed to by `p` and returns the root's state label. State zero labels unmatched trees. The trees may be corrupt or merely inconsistent with the grammar.

The simpler `burm_state` is `burm_label` without the code to traverse the tree and to read and write its fields. It may be used to integrate labelling into user-supplied

[1] BURG does not require that the input grammar be in normal form (§4.3.1).

traversal code. A typical signature is

```
extern int burm_state(int op, int leftstate, int rightstate);
```

It accepts an external symbol number for a node and the state labels for the node's left and right children. It returns the state label to assign to that node. For unary operators, the last argument is ignored; for leaves, the last two arguments are ignored. In general, BURG generates a `burm_state` that accepts the maximum number of child states required by the input grammar. For example, if the grammar includes no binary operators, then `burm_state` will have the signature

```
extern int burm_state(int op, int leftstate);
```

This feature is included to permit future expansion to operators with more than two children.

The user must write the reducer, but BURM writes code and data that help. Primary is

```
extern int burm_rule(int state, int goalnt);
```

which accepts a tree's state label and a goal non-terminal and returns the external rule number of a rule. The rule will have matched the tree and have the goal non-terminal on the left-hand side; `burm_rule` returns zero when the tree labelled with the given state did not match the goal non-terminal. For the initial, root-level call, `goalnt` must be one, and BURM exports an array that identifies the values for nested calls:

```
extern short *burm_nts[] = { ... };
```

is an array indexed by external rule numbers. Each element points to a zero-terminated vector of short integers, which encode the goal non-terminals for that rule's pattern,

left-to-right. The user needs only these two externals to write a complete reducer, but a third external simplifies some applications:

```
extern NODEPTR_TYPE *burm_kids(NODEPTR_TYPE p,
                               int eruleno,
                               NODEPTR_TYPE kids[]);
```

accepts the address of a tree p, an external rule number, and an empty vector of pointers to trees. The procedure assumes that p matched the given rule, and it fills in the vector with the subtrees (in the sense described above) of p that must be reduced recursively. kids is returned. It is not zero-terminated.

The simple user code below labels and then fully reduces a subject tree; the reducer prints the tree cover. burm_string is defined below.

```
parse(NODEPTR_TYPE p) {
  burm_label(p);   /* label the tree */
  reduce(p, 1, 0);  /* and reduce it */
}

reduce(NODEPTR_TYPE p, int goalnt, int indent) {
  int eruleno = burm_rule(STATE_LABEL(p), goalnt); /* matching rule # */
  short *nts = burm_nts[eruleno];       /* subtree goal non-terminals */
  NODEPTR_TYPE kids[10];                          /* subtree pointers */
  int i;

  for (i = 0; i < indent; i++)
    printf(".");                                  /* print indented ... */
  printf("%s\n", burm_string[eruleno]);            /* ... text of rule */
  burm_kids(p, eruleno, kids);          /* initialize subtree pointers */
  for (i = 0; nts[i]; i++)          /* traverse subtrees left-to-right */
    reduce(kids[i], nts[i], indent+1);  /* and print them recursively */
}
```

The reducer may recursively traverse subtrees in any order, and it may interleave arbitrary semantic actions with recursive traversals. Multiple reducers may be written, to implement multi-pass algorithms or independent single-pass algorithms.

For each non-terminal x, BURG emits a preprocessor directive to equate `burm_x_NT` with x's integral encoding. It also defines a macro `burm_x_rule(a)` that is equivalent to `burm_rule(a,x)`. For the grammar in Figure A.1, BURG emits

```
#define burm_reg_NT 1
#define burm_con_NT 2
#define burm_addr_NT 3
#define burm_reg_rule(a) ...
#define burm_con_rule(a) ...
#define burm_addr_rule(a) ...
```

Such symbols are visible only to the code after the second "`%%`". If the symbols `burm_x_NT` are needed elsewhere, extract them from the BURM source.

The `-I` option directs BURG to emit an encoding of the input that may help the user produce diagnostics. The vectors

```
extern char *burm_opname[];
extern char burm_arity[];
```

hold the name and number of children, respectively, for each terminal. They are indexed by the terminal's external symbol number. The vectors

```
extern char *burm_string[];
extern short burm_cost[][4];
```

hold the text and cost vector for each rule. They are indexed by the external rule number. The zero-terminated vector

```
extern char *burm_ntname[];
```

is indexed by burm_x_NT and holds the name of non-terminal x. Finally, the procedures

```
extern int burm_op_label(NODEPTR_TYPE p);
extern int burm_state_label(NODEPTR_TYPE p);
extern NODEPTR_TYPE burm_child(NODEPTR_TYPE p, int index);
```

are callable versions of user-defined configuration macros. `burm_child(p,0)` implements `LEFT_CHILD(p)`, and `burm_child(p,1)` implements `RIGHT_CHILD(p)`. A sample use is the grammar-independent expression `burm_opname[burm_op_label(p)]`, which yields the textual name for the operator in the tree node pointed to by `p`.

A complete tree parser can be assembled from just `burm_state`, `burm_rule`, and `burm_nts`, which use none of the configuration section except `PANIC`. The generated routines that use the rest of the configuration section are compiled only if the configuration section defines `STATE_LABEL`, so they can be omitted if the user prefers to hide the tree structure from BURM. This course may be wise if, say, the tree structure is defined in a large header file with symbols that might collide with BURM's.

BURM selects an optimal parse without doing dynamic programming at compile time [AJ76]. Instead, BURG does the dynamic programming at compile-compile time, as it builds BURM. Consequently, BURM parses quickly. Similar labellers have taken as few as 15 instructions per node, and reducers as few as 35 per node visited [FH91c].

A.4 Debugging

BURM invokes `PANIC` when an error prevents it from proceeding. `PANIC` has the same signature as `printf`. It should pass its arguments to `printf` if diagnostics are desired and then either abort (say via `exit`) or recover (say via `longjmp`). If it returns, BURM aborts. Some errors are not caught.

```
%term Const=17 RedFetch=20 GreenFetch=21 Plus=22
%%
reg: GreenFetch(green_reg) = 10 (0);
reg: RedFetch(red_reg) = 11 (0);

green_reg: Const = 20 (0);
green_reg: Plus(green_reg,green_reg) = 21 (1);

red_reg: Const = 30 (0);
red_reg: Plus(red_reg,red_reg) = 31 (2);
```

Figure A.3: A Diverging Tree Grammar

BURG assumes a robust preprocessor, so it omits full consistency checking and error recovery. BURG constructs a set of states using the algorithm of Chapter 4. BURG considers all possible trees generated by the tree grammar and summarizes infinite sets of trees with finite sets. The summary records the cost of those trees but actually manipulates the differences in costs between viable alternatives using a dynamic programming algorithm.

Some grammars derive trees whose optimal parses depend on arbitrarily distant data. When this happens, BURG and the tree grammar *cost diverge*, and BURG attempts to build an infinite set of states; it first thrashes and ultimately exhausts memory and exits.[2] For example, the tree grammar in Figure A.3 diverges, since non-terminals `green_reg` and `red_reg` derive identical infinite trees with different costs. If the cost of rule 31 is changed to 1, then the grammar does not diverge.

Practical tree grammars describing instruction selection do not cost-diverge because infinite trees are derived from non-terminals that model temporary registers. Machines can move data between different types of registers for a small bounded cost, and the

[2]The `-c` option sets a threshold to abort processing if the grammar appears to diverge.

rules for these instructions prevent divergence. For example, if Figure A.3 included rules to move data between red and green registers, the grammar would not diverge. If a bonafide machine grammar appears to make BURG loop, try a host with more memory. To apply BURG to problems other than instruction selection, be prepared to consult the literature on cost-divergence [PL88].

A.5 Running BURG

BURG reads a tree grammar and writes a BURM in C. BURM can be compiled by itself or included in another file. When suitably named with the `-p` option, disjoint instances of BURM should link together without name conflicts. The command:

`burg` [*arguments*] [*file*]

invokes BURG. If a *file* is named, BURG expects its grammar there; otherwise it reads the standard input. The options include:

`-c` N Abort if any relative cost exceeds N, which keeps BURG from looping on diverging grammars. [PLG88, Hen89, BDB90, Pro92] explain relative costs.

`-d` Report a few statistics and flag unused rules and terminals.

`-o` *file* Write parser into *file*. Otherwise it writes to the standard output.

`-p` *prefix* Start exported names with *prefix*. The default is `burm`.

`-t` Generates smaller tables faster, but all goal non-terminals passed to `burm_rule` must come from an appropriate `burm_nts`. Using `burm_`x`_NT` instead may give unpredictable results.

`-I` Emit code for `burm_arity`, `burm_child`, `burm_cost`, `burm_ntname`, `burm_op_label`, `burm_opname`, `burm_state_label`, and `burm_string`.

`-O` N Change the principal cost to N. Elements of each cost vector are numbered from zero.

`-=` Compare costs lexicographically, using all costs in the given order. This option slows BURG and may produce a larger parser. Increases range from small to astronomical.

A.6 Acknowledgements

The first BURG was adapted by the second author from his CODEGEN package, which was developed at the University of Washington with partial support from NSF Grant CCR-88-01806. It was unbundled from CODEGEN with the support of Tera Computer. We wrote the current BURG with the support of NSF grant CCR-8908355. The interface, documentation, and testing involved all three authors.

Comments from a large group at the 1991 Dagstuhl Seminar on Code Generation improved BURG's interface. Robert Giegerich and Susan Graham organized the workshop, and the International Conference and Research Center for Computer Science, Schloss Dagstuhl, provided an ideal environment for such collaboration. Beta-testers included Helmut Emmelmann, Dave Hanson, John Hauser, Hugh Redelmeier, and Bill Waite.

Bibliography

[AG85] Alfred V. Aho and Mahedevan Ganapathi. Efficient tree pattern matching: An aid to code generation. In *Proceedings of the 12th Annual Symposium on Principles of Programming Languages*, pages 334–340, January 1985.

[AGT89] Alfred V. Aho, Mahedevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[AJ76] A. V. Aho and S. C. Johnson. Optimal code generation for expressions trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[AJU77] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *Journal of the ACM*, 24(1):146–160, January 1977.

[App87] Andrew W. Appel. Concise specifications of locally optimal code generators. Technical Report CS-TR-080-87, Princeton University, Dept. of Computer Science, Princeton, New Jersey, February 1987.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *Proceedings of the SIGPLAN '89*

Conference on Programming Language Design and Implementation, pages 275–284, 1989.

[BDB90] A. Balachandran, D. M. Dhamdhere, and S. Biswas. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages*, 15(3):127–140, 1990.

[Bea74] J. C. Beatty. Register assignment algorithm for generation of highly optimized object code. *IBM Journal of Research and Development*, 18(1):20–39, January 1974.

[BEH91] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for riscs. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[BJR89] David Bernstein, Jeffrey M. Jaffe, and Michael Rodeh. Scheduling arithmetic and load operations in parallel with no spilling. *SIAM Journal on Computing*, 18(6):1098–1127, December 1989.

[BL92] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[BMW87] Jürgen Börstler, Ulrich Mönche, and Reinhard Wilhelm. Table compression for tree automata. Technical Report Aachener Informatik-Berichte 87–12, Fachgruppe Informatik, Aachen, Fed. Rep. of Germany, 1987.

[BPR84] David Bernstein, Ron Y. Pinter, and Michael Rodeh. Optimal scheduling of arithmetic operations in parallel with memory access. In *Proceedings of the 12th Annual Symposium on Principles of Programming Languages*, pages 325–333, January 1984.

[CAC+81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cooke, M. E. Hopkins, and P. W. Markstein. Register allocation via graph coloring. *Computer Languages*, 6:47–57, January 1981.

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, January 1990.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 98–101, June 1982.

[Cha87] David R. Chase. An improvement to bottom-up tree pattern matching. In *Proceedings of the 14th Annual Symposium on Principles of Programming Languages*, pages 168–177, 1987.

[CK91] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 192–203, June 1991.

[Cof76] E. G. Coffman, Jr., editor. *Computer and Job-Shop Scheduling Theory.* John Wiley and Sons, 1976.

[ESL89] Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. BEG—a generator for efficient back ends. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 227–237, 1989.

[FH91a] Christopher W. Fraser and David R. Hanson. A code generation interface for ANSI C. *Software—Practice and Experience*, 21(9):963–988, September 1991.

[FH91b] Christopher W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10), October 1991.

[FH91c] Christopher W. Fraser and Robert R. Henry. Hard-coding bottom-up code generation tables to save time and space. *Software—Practice and Experience*, 21(1):1–12, January 1991.

[FHP91] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1991.

[FL88] Charles N. Fischer and Richard J. Leblanc, Jr. *Crafting a Compiler.* Benjamin/Cummings, Menlo Park, California, 1988.

[Fre74] R. A. Freiburghouse. Register allocation via usage counts. *Communications of the ACM*, 17(11), November 1974.

[GH88] James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. In *International Conference on Supercomputing*, July 1988.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[GM86] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 11–16, 1986.

[Han90] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice and Experience*, 20(1):5–12, January 1990.

[HC86] Philip J. Hatcher and Thomas W. Christopher. High-quality code generation via bottom-up tree pattern matching. In *Proceedings of the 13th Annual Symposium on Principles of Programming Languages*, pages 119–130, 1986.

[Hen89] Robert R. Henry. Encoding optimal pattern selection in a table-driven bottom-up tree-pattern matcher. Technical Report 89-02-04, University of Washington, 1989.

[HFG89] Wei-Chung Hsu, Charles N. Fischer, and James R. Goodman. On the minimization of loads/stores in local register allocation. *IEEE Transactions on Software Engineering*, 15(10):1252–1260, 1989.

[HG82] John L. Hennessy and Thomas R. Gross. Code generation and reorganization in the presence of pipeline constraints. In *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*, pages 120–127, 1982.

[HG83] John L. Hennessy and Thomas R. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[HO82] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[Hu61] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[KFP92] Steven M. Kurlander, Charles N. Fischer, and Todd A. Proebsting. Extended delayed-load scheduling. Technical report (in preparation), University of Wisconsin, 1992.

[Kro75] H. Kron. *Tree Templates and Subtree Transformational Grammars.* PhD thesis, University of California, Santa Cruz, 1975.

[LH86] J. R. Larus and P. N. Hilfinger. Register allocation in the SPUR lisp compiler. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 255–263, 1986.

[LLM+87] Eugene Lawler, Jan Karel Lenstra, Charles Martel, Barbara Simons, and Larry Stockmeyer. Pipeline scheduling: A survey. Computer science research report, IBM Research Division, 1987.

[Mor91] W. G. Morris. CCG: A prototype coagulating code generator. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 45–58, 1991.

[PF91] Todd A. Proebsting and Charles N. Fischer. Linear-time optimal code scheduling for delayed-load architectures. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[PF92] Todd A. Proebsting and Charles N. Fischer. Probabilistic register allocation. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992.

[PH90] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Palo Alto, California, 1990.

[PL88] Eduardo Pelegri-Llopart. *Rewrite Systems, Pattern Matching, and Code Generation.* Phd Thesis, Technical Report UCB/CSD 88/423, Computer Science Division, University of California, Berkeley, 1988.

[PLG88] Eduardo Pelegri-Llopart and Susan L. Graham. Optimal code generation for expression trees: An application of BURS theory. In *Proceedings of the 15th Annual Symposium on Principles of Programming Languages*, pages 294–308, 1988.

[Pro92] Todd A. Proebsting. Simple and efficient burs table generation. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992.

[PS90] Krishna Palem and Barbara Simons. Scheduling time-critical instructions on RISC machines. In *Proceedings of the 17th Annual Symposium on Principles of Programming Languages*, pages 270–280, 1990.

[SU70] Ravi Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, October 1970.

[Wai91] William Waite. Personal communication. (Electronic mail), December 1991.

[Wal86] David W. Wall. Global register allocation at link time. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 264–275, 1986.

[War90] H. S. Warren, Jr. Instruction scheduling for the IBM RISC system/6000 processor. *IBM Journal of Research and Development*, 34(1):85–92, 1990.