CMU-CS-84-137

Cost-Minimization in Register Assignment for Retargetable Compilers

Andrew Reiner

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

8 June 1984

Abstract

A method of register assignment in optimizing retargetable compilers is described. The design of the PACK register assignment system is presented, and it is shown how PACK functions in compilers built with the Production Quality Compiler Compiler technology.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

List of Figures

1. Introduction

While its clearly desirable to build fast compilers that generate optimized code, its less obvious how to construct them cheaply and quickly.

Compilers generate good code by exploiting some features of the target machine (e.g., special-purpose registers or powerful addressing modes) and avoiding others (such as memory accesses that require preloading of base registers). This evaluation of the machine can only be done if it's properties—and their benefits or costs—are known by the compiler. Thus, a compiler for some machine X is an "expert" on generating code for X, by virtue of the knowledge built into it about X. Modifying the compiler to generate code for some other machine Y (a process called *retargeting*), requires building in optimization-type information about Y.

The relative complexity of optimizing compilers has caused their construction (by building from scratch or by retargeting an existing one) to be an expensive proposition. The cost is largely due to the fact that the set of procedures and information needed to correctly perform optimizations is usually ill-defined. Designing the right optimizations for a machine, implementing them correctly, and validating that the compiler produces code that is both correct and of high quality all add additional expense to the task.

Efforts to help keep this expense in hand have led to the development of *retargetable* compilers that can be tailored to a new target machine with a minimum of effort. One promising approach has attacked the problem by consciously "designing out" machine information: instead of having it strewn ad-hoc throughout the compiler's algorithms, the knowledge is localized in a formal, well-defined form, where it is accessed as needed during compilation. The optimization algorithms become machine independent, because all of their knowledge about the machine structure must be drawn from the formal description. This avoids one of the potentially expensive retargeting tasks—the redesign of algorithms that are unusable because they embody assumptions about the old machine.

There are other advantages from the retargeter's viewpoint. First, it is not necessary to hunt out and change machine-specific information, since all the information needed by the compiler is localized. Secondly, there exist rules which dictate what information is needed, and in what form. The person who assembles it does not have an intimate knowledge of the compiler organization, or even be knowledgeable about compiler technology.

In this paper, we'll examine how this particular retargetability method is used in the design of PACK, the *storage packing* section of an optimizing compiler.

Packing (which is also referred to as storage assignment or register assignment) is one of the final compiler tasks involved in the processing of program variables and compiler-generated temporaries.

The PACK phase *binds* each program variable[1] to a location or sequence of locations in storage. A binding indicates where in storage the contents of the variable will reside.

PACK is only responsible for finding bindings for variables. Earlier phases have looked after preliminary tasks such as lifetime analysis and mapping between language types and machine representations. Most of the information about variables that Pack needs is supplied to it by these phases.

1.1 What makes a good binding?

PACK can be simply defined as a phase that starts with a set of information relating to program variables, and generates a set of bindings, one for each variable. These bindings must be *legal*, in terms of program correctness, and they should whenever possible aid in making the compiled program small and fast.

1.1.1 Legality

The first consideration of a packing algorithm is, reasonably enough, that the bindings it produces do not in some way invalidate the correctness of the object program. For storage packing, the criterion for correctness is the *disjoint-lifetime* constraint. Each variable has a *lifetime*, consisting of those periods in the program when it contains a value that will be used at a later time. If the following code fragment contains the only occurrances of a variable A in a program

```
A := B + C;

. . .

D := B * A;
```

then the lifetime of A spans the period of execution between (and including) these two statements. In a correct packing, no two variables with overlapping lifetimes may occupy the same storage location[2]. Variables overlap if there exists any point in the program that is contained in the lifetime spans of both.

---

[1] Hereafter, the term "variable" will mean "variable or compiler temporary."

[2] Unless they contain the same value during the periods of overlap.

1.1.2 Quality

Code quality is commonly measured by program size and execution speed. The *cost* of a compilation decision (e.g., a variable binding) is therefore measured by how it affects the program's size and speed.

The treatment of storage assignment set out in this paper deals exclusively with execution costs, since good execution performance usually implies good size statistics. However, the techniques are equally applicable to an optimization model based on program size minimization.

In the scheme of compilation that will be developed here, variable binding is done as a separate operation prior to code generation. This differs from many designs in which binding and code generation are done in a co-routine fashion. Thus, the bindings constrain the choices of the code generation phase. If the bindings are good, then it is easy to emit good code. If the bindings are poor, then code quality will suffer.

There are two basic ways that a good set of bindings can improve code quality: By allowing the code generator to exploit inexpensive instructions, or by minimizing the need for location-to-location copies of data items. Illustrations of each will follow.

1.1.2.1 Constraints by operand locations on instruction selection

There are two situations where the variable bindings have a direct affect on the selection of instructions by the code generator. First, consider how the 2-address addition

```
A := A + B
```

might be computed on a PDP-10. The cost of computation depends on whether the operands are bound to registers or memory. If both are bound to registers, then the code generator can emit

```
ADD Loc_A, Loc_B
```

"Loc_X" is the machine location where variable X has been bound. If for some reason A is bound to memory, then a more expensive instruction must be used:

```
ADDM Loc_B, Loc_A
```

This instruction adds a register value B to a memory value A, and stores the result in A. During the packing process, the cost difference of these instructions should be considered when determining locations for A and B. If B has already been bound to a register, and A is yet to be bound, then part of the cost of binding A to a memory location (instead of a register) is the difference in cost between the ADD and the ADDM.

The second case where operand location affects instruction selection arises on machines with both two- and three-operand instructions, such as the Vax.

On the Vax, an expression such as

```
A := B + C
```

can be coded in two different ways. The one selected depends on operand bindings. If A, B, and C are bound to different locations, then a three-operand instruction is selected:

```
ADDL3 Loc_B, Loc_C, Loc_A
```

However, if lifetimes permit, the destination operands A may be bound to the same location as one of the source operands—for instance, B. Since Loc_A is the same as Loc_B, a cheaper two-address instruction can be used:

```
ADDL2 Loc_C, Loc_A
```

Thus, PACK must be aware of the potential savings when A and B (or A and C) are bound to the same location. This is called a *local preference* relation. The local preference relation will be discussed at greater length later.

1.1.2.2 Data move minimizations

The issue in the optimization just discussed was how to arrange things so as to use a fast instruction. In contrast, data move mimization is concerned with binding variables so as to eliminate the need for additional instructions to perform data transfers between storage locations.

Consider once again how the coding of an expression on the PDP-10 is affected by operand location. Suppose the operands of the expression

```
A := A + B
```

are *both* bound to memory locations. The PDP-10 has no instruction that adds the contents of two memory locations, so one of the two operands must be preloaded into a temporary register. This data move—the loading of an operand into a register—would have been avoided if PACK had the foresight to bind one of operands to a register in the first place.

The *global preference* problem is a second example of how good binding can save a data move. There are situations where two variables must reside in the same storage location. Thus, if Pack does not bind them together, then the code generator will be forced to generate an explicit data move instruction. These two variables are said to be associated by the global preference relation.

1.2 The General Strategy

In the examples above, it was shown how impediments to program efficiency could be avoided by proper bindings of variables. The process of choosing a good binding for a variable *X* can be divided into two steps. The first step is to find which storage area (e.g., registers or memory) is best cost-wise. The second step finds the best *location* within the area.

The search for a suitable storage area is called *storage class determination*. Most variables can legally be packed to any one of a number of storage types. PACK decides on the one to use by ranking the desirability of the types cost-wise, with the most desirable having the smallest cost. PACK is able to evalute the efficiency loss that will be accrued if the variable is not bound there. This allows the intellegent allocation of storage such as general registers, which is desired by a large number of variables, but contains a limited number of locations. If not every variable that desires a register binding can be accomodated, then priority will be given to those with large penalty costs.

The second step, called *location selection*, evaluates the desirability of each location according to a set of criteria, and selects the one with the lowest cost. If, for example, one of the locations contains a variable that is preference-linked to *X*, the variable being bound, then that location is more attractive, because binding *X* there will save a data move.

This paper will chiefly focus on a method for storage class determination, and on the way in which a packing algorithm can be retargeted. Chapter 2 lays down the groundwork, discussing the overall structure of PACK. Chapter 3 focuses on the storage class determination process and the machine information that supports it. Chapter 4 outlines location selection, and chapter 5 discusses performance and retargetability topics.

The coding examples that appear in later chapters use the instruction sets of DEC's PDP10 and Vax computers. The Vax instructions are written with the opcode first, followed by the source operands, and finally the destination operand, e.g.,

```
ADDL3 src1, src2, dst
```

The Vax instructions that appear here will have two or three operands.

All PDP10 instructions have two operands, and in most, the destination operand appears first:

```
ADD dst, src
```

However, instructions whose opcodes end in "M" store results to memory, and are written with the operands in the reverse order:

```
ADDM src, dst
```

2. The PACK Storage Packing Phase

The business of storage assignment is beguilingly simple to state—assign variables to storage in such a way that the size and/or execution speed of the program is minimized.

Modest as the task seems, workers in the area found long ago that the problem is NP-complete, and that no tractable algorithm was likely to be found to optimally solve the general register-assignment problem, given that lifetime overlaps are not allowed. Attention was then turned toward methods that yielded approximate solutions: the assignment of storage is not perfect, but it comes close enough.

One of the more widely used techniques models the register assignment problem as a weighted graph-coloring task (another NP-complete problem), and finds approximate solutions for it. Variables are represented as graph nodes, and the connecting edges represent the lifetime conflict relation. Each node has a numerical weight that indicates the relative desirability of storing that variable in a register. The task is to find an N-coloring of the graph. An N-coloring assigns each node one of N colors (or integers) such that no two nodes connected by an edge are colored the same. If N is the number of registers available, than such a coloring corresponds to an assignment of the variables to the N registers such that no conflicting variables share the same one. If no complete coloring can be found, then a partial coloring must be used instead. In a partial coloring, some of the nodes are left uncolored. Uncolored nodes correspond to variables that will be assigned to memory instead of a register. Selecting which nodes to leave uncolored is done with the node weights: the sum of the uncolored node weights should be as small as possible. Thus, a minimum of nodes are left uncolored, and those that are should be as "unimportant" as possible. The general scheme is shown in Figure 2-1. Leverett [Leverett 81] surveys some designs which use this scheme, and also examines some other basic models of the task.

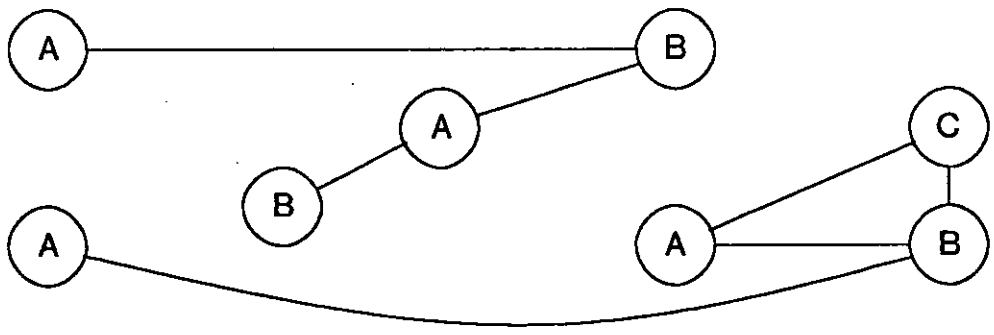


Figure 2-1: A graph coloring. The arcs denote lifetime conflicts, and the letters represent colors

PACK's task is a bit harder than this simple version of the graph-coloring problem, because it may have to deal with an arbitrary number of storage types, rather than just two. It must also handle a second class of TN interdependency, in addition to the lifetime conflict arcs—the preference relations.

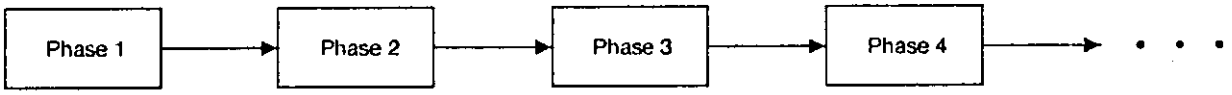
2.1 PQCC Preliminaries

PACK was originally designed for use in compilers built with the Production-Quality Compiler Compiler (PQCC), an experimental compiler-writing system developed at Carnegie-Mellon University. To understand PACK's layout, one must become familiar with some of the structure of these PQCC-constructed compilers, which are called Production Quality Compilers (PQCs). Before discussing PACK proper, we'll look at the ways that the PQC represents the source program during compilation, and it's storage description notation. This will also touch on two phases of importance to PACK—LTN, which generates input to PACK, and CODE, which uses PACK's output.

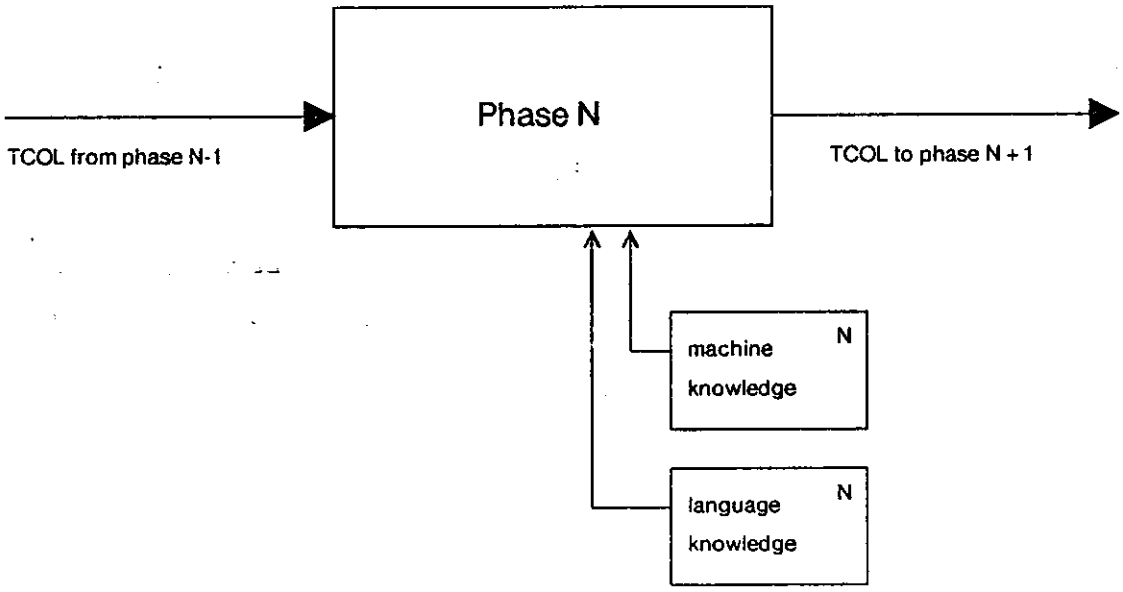
This quick introduction to the PQCC system will be suitable preparation for understanding PACK. However, a proper overview of the methods developed in PQCC is obtainable from a number of survey and technical publications. [Leverett 80] gives a complete introduction to the techniques of PQCC-style compiler construction, while [Wulf 80] is a shorter, more accessible survey that examines PQCC as a problem-solving system. Of particular interest are Leverett's studies of register allocation ([Leverett 81], [Leverett 82]). These examine techniques that are the intellectual forebears of many of the idea presented in this paper. These works have recently been combined and published in book form ([Leverett 83]).

2.1.1 General Structure of the PQC

In a PQCC-style compiler, schematically shown in figure 2-2a, the work is divided among a set of discrete, sequentially executed processes called phases. Each phase has a clearly defined responsibility, and is the locus of the procedural knowledge needed to discharge it. This knowledge, however, does not include information about the source language or target machine. Instead, each phase is given this information in declarative form, e.g., by tables (figure 2-2b). Since each phase has a unique function, it is natural that each phase requires different knowledge about the language and machine to correctly carry out its task. During compiler construction, information from the common descriptions supplied by the compiler writer are distilled into phase-specific tables that contain only what is needed for a particular phase's operation. Thus, a phase *N* has descriptions of the target machine and source language that are tailored to the phase's operation (figure 2-2b). There are some phases whose operation does not depend on properties of the source language, and thus have no need of a language description input. PACK is one of these phases.



(a)



(b)

Figure 2-2:
(a) Schematic of a PQCC-style compiler
(b) Phase structure and communication

Many of the phases prior to PACK are involved in collecting information that will be used for storage assignment (e.g., lifetime information). The most important of these is LTN, which is responsible for the creation of temporaries and which creates preference links. LTN executes immediately prior to PACK.

The phase following PACK is the code selection phase, CODE. Since PACK runs before CODE, all TNs are bound by PACK before CODE begins execution. Although the CODE phase may make some minor adjustments to the packing decisions (for example, by doing *VeryTemp* allocation, a form of register spilling), it generally has to make do with the bindings it gets from PACK. If PACK makes a bad choice, then CODE may have to emit more costly (slower) instructions.

2.1.2 The Intermediate Representation

In figure 2-2, the arrows indicate the flow of program information from one phase to the next. This consists of the source program (in it's intermediate state), and the accumulated knowledge about the program that was gathered by prior phases for the benefit of future phases (e.g., flow analysis information). All of this data is represented in a language called TCOL (for *T*ree *CO*mmon *L*anguage). TCOL is a tree-structured operator langauge. Nodes consist of one of a variety of operators, whose operands are subtrees or leaves. Figure 2-3 shows the representation of a simple arithmetic expression in TCOL.

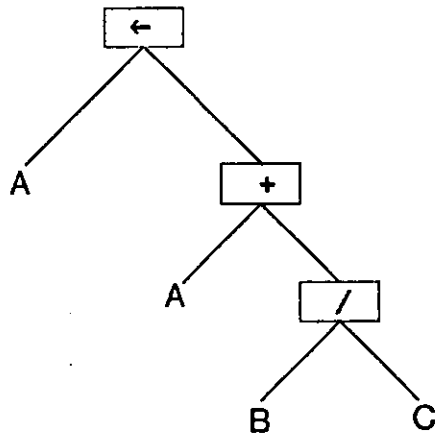


Figure 2-3: A TCOL representation of "A := A + (B/C)"

In addition to simple arithmetic expressions like this, TCOL trees also contain other objects that hold information about the state of the compilation (e.g., the feasibility and desirability of a particular optimization). TCOL allows the compiler to store this "bookkeeping" information in amongst the program representation.

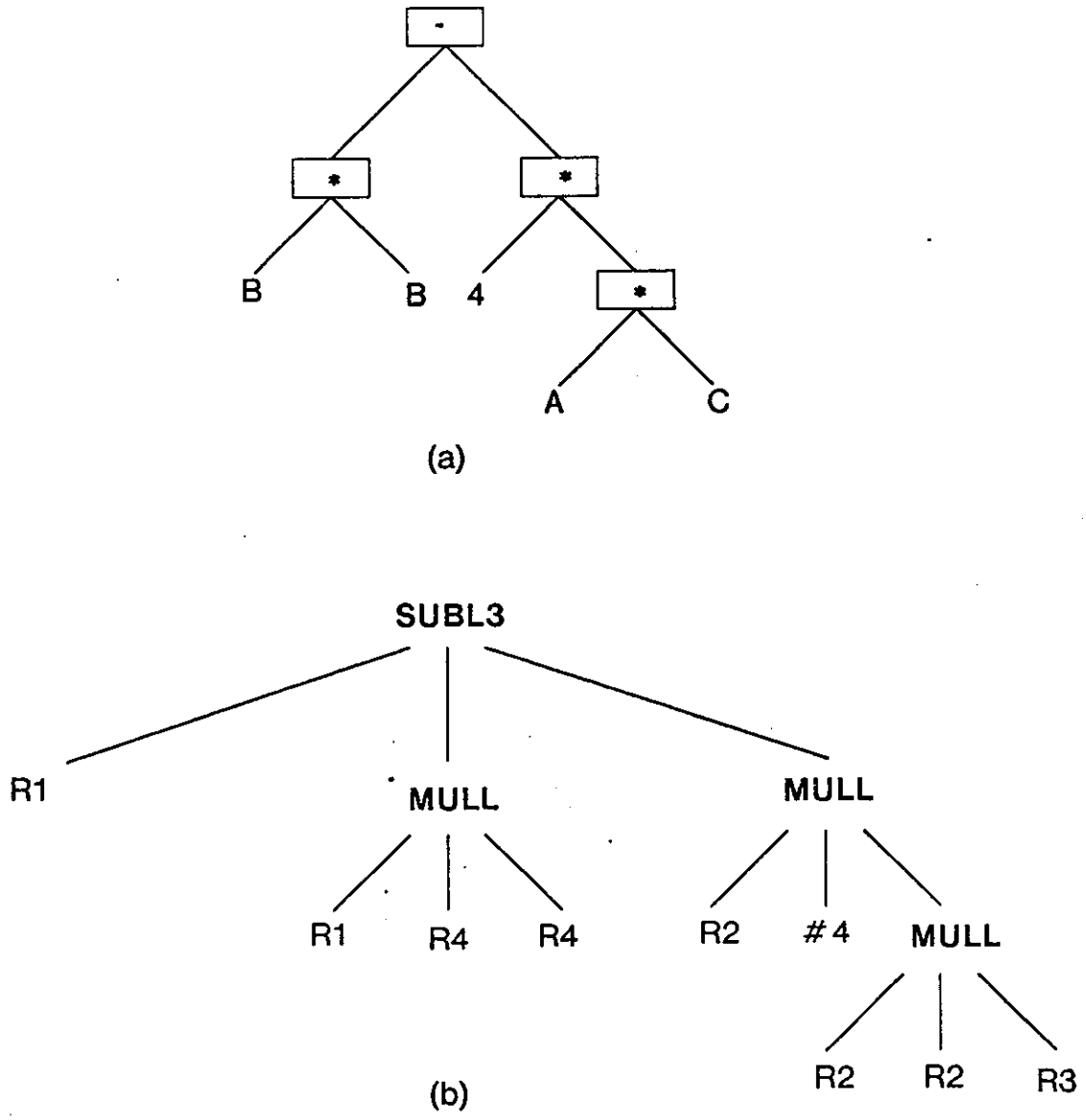


Figure 2-4: The TCOL representation of $B^2 - 4AC$, as it might appear at two points in the compilation

The PQCC phases can be viewed as a set of "tree modifiers", which each receive one tree as input, and emit a new tree as output. A phase might append new information to the tree, or replace a subtree component with a new subtree that contains new types of operators and operands. The initial TCOL representation of a program at the start of the phase pipeline will be quite different from the way it looks at the end. This is illustrated in figure 2-4, which shows a tree fragment during the initial and final stages of the compilation. Fig. 2-4a shows how the tree might look shortly after parsing. The operators and operands correspond closely to their representation in the source program. The

second diagram shows the tree after code generation, right before it is "linearized" into object-program form. Here, the operators are Vax mnemonics, and the operands denote literals and general register indices.

PACK uses a format of TCOL that reflects its status as a language-independent phase that is positioned towards the end of the phase sequence. The tree operands corresponding to program variables do not contain any source langauge semantics, such as language type information, The operators, on the other hand, express notions such as data length that correspond directly to properties of the target machine. The following section describes the tree format in detail.

This short discussion of TCOL has avoided the features and issues that make intermediate representations a central issue in compiler construction technology. A proper treatment of TCOL and its descendents can be found in [Brosgol 80] and [Goos 81].

2.2 Input and Output Tree Formats

2.2.1 The PACK Input Tree

Figure 2-5 shows a representative input tree. The interior nodes are *Virtual Operators* (Vops). Vops are one of the intermediate forms that lie on the representational path between source language operators and target machine instructions. Vops appear in the tree during the latter stages of compilation, and are replaced in the code-selection phase by machine instructions or instruction sequences.

A Vop expresses what is currently known about the machine instruction (or sequence) that may eventually be generated in it's place. It indicates the general type of computation, and the type and length of the operands. The Vop identifier consists of three fields:

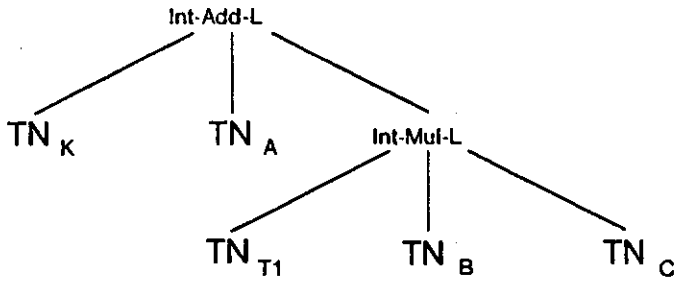
DATATYPE_OPCLASS_LENGTH

where

- DATATYPE is an operand type: Real, Integer, Bitstring, etc.
- OPCLASS is a general operation classname: Addition, Multiplication, Shift, etc.
- LENGTH is an operand length: Byte, Halfword, etc.

It should be apparent that the information encoded in a Vop is highly machine-specific. Vop definitions are thus dependent on the target machine. As a consequence, compilers with different target machines use TCOLs that differ in the values of their Vop sets. This is an example of a

machine-dependent compiler component that must be respecified during retargeting[3]. Although Vops are machine-dependent, they are not true machine instructions, nor is there a simple one-to-one correspondence between the two.



K := A + (B * C)

Figure 2-5: Examples of PACK's TCOL input

The leaves of the input tree are *Tempnames* (TNs), objects that represent variables and compiler temporaries. Each variable has a distinct TN, which contains items of interest to PACK, including lifetime information, and *usage counts*, which are an indication of how frequently the variable is accessed. The object represented by a TN may be a compound structure (e.g., an array or record) or a scalar. Compound structures are generally bound to memory, and so are not a major issue in the allocation of scarce, high speed registers, which is our major concern. The specialized techniques for binding large structures to memory will be avoided here, but may be found in [Fabri 79].

2.2.1.1 Input Tree Syntax

The input tree format follows a few well-defined rules. In general, each Vop type has a set number of operands. This discussion of PACK will be illustrated by examples drawn from binary arithmetic, using a form of "three-address" type Vops described in the next paragraph. However, bear in mind that what follows is generally applicable to expression trees containing Vops with more or fewer operands.

A three-address Vop has three operands—one *destination* operand, and two *source* operands. These Vops naturally express arithmetic operations where all three operands are distinct TNs. However, two-address computations can be expressed by having the destination operand appear as one of the sources.

---

[3] TCOL undergoes other changes during retargeting, but they are not relevant to this discussion.

In complex or nested computations, a source operand of the "root" Vop will be obtained from the result of some sub-computation. In the tree, the sub-expression Vop is placed below the root Vop in a position corresponding to one of the root's source operands, as is shown in figure 2-5. This is interpreted as meaning that the subexpression destination TN serves as the source TN one level up.

2.2.2 The Output Tree

The only modification PACK makes to the tree is the addition of binding information to each TN. This specifies the type of storage, and the location or sequence of locations that have been assigned.

2.3 Describing the Structure of Storage

There exists a simple notation for representing the fundamental characteristics of the target architecture's storage layout. The technique of representing this knowledge is an issue, since it must be general enough to describe the storage setups of any number of potential target machines.

Storage bases and *storage classes* are the record structures that hold information about storage. Storage bases describe the basic formats of storage. Storage classes add an additional level of organization, representing storage as element sequences whose members share a common size specification and address constraint.

There exists one storage base (abbreviated "SB") for each of the common areas of storage, such as general registers, memory, etc. In general, there will be one SB for memory, and one for each distinct type of register. For instance, an SB description of a machine with index registers and accumulators contains three SBs (one for memory, two for the registers), while a machine having general registers is described by two SBs.

An SB describes a storage type by indicating the number of addressable units contained in it, and the size of each unit. For example, the Vax's sixteen registers are described by an SB **SB_Reg**:

SB_Reg: Num_Elements = 16, Element_Size = 32

Individual elements are represented by a subscript, e.g.,

SB_Reg[4]

Subscripts run from 0 to Num_Elements – 1.

Straightforward as this definition is, one must frequently use care when defining storage bases, for there is usually more than one legal SB definition for a particular storage area. This problem arises when an architecture supports multiple data lengths in the same storage area. For instance, the contents of Vax memory may be addressed as bytes, words, or longwords. The competing SB definitions are:

- Num_Elements = 2↑32, Element_Size = 8
- Num_Elements = 2↑31, Element_Size = 16
- Num_Elements = 2↑30, Element_Size = 32

The rule is to select the definition that has the *smallest element size*. In this example, the proper definition is the first.

Many computer architectures support stacks, and it is convenient to model them as storage bases, rather than as a constituent of main memory. Stacks differ in many ways from other storage areas—the number of elements changes during program execution, for example. These non-standard features are masked by other compiler phases, and PACK can handle stacks as it would any other SB.

While SBs describe the basic properties of storage, they are too simple to directly categorize the way in which storage is used by the machine instructions, since instructions handle data that may be larger than the smallest addressable unit. For instance, although a single Vax register handles only four bytes, floating point instructions may have operands that are eight or sixteen bytes long. Such operands require (respectively) two and four contiguous registers.

Another factor to consider is that practically every architecture has instructions whose operands are restricted to reside in special registers. A divide instruction might require that the result be evaluated in an odd-numbered register, or a character-processing instruction might require its operand to be in a designated register (e.g., general register 4).

The common problem here is that instructions look at storage as something more than a simple sequence of atomic units. An instruction operand might access a larger chunk of storage. It also might restrict itself to a certain subset of elements or chunks.

A storage class (SC) is a refinement of a storage base. An SC X is defined by three properties:

- A Storage Base specification X_{SB} (also called the *root* SB).
- An SC element size specification X_S.
- An SC restriction specification X_R.

X is a sequence of elements, each of which consists of a contiguous sequence of X_S SB elements. *Which* sequences are members of the SC depend on the restriction, which is an integer sequence that indicates the SB addresses that are valid starting positions.

To illustrate the properties of storage classes, consider the register-pair SC for the Vax. The Vax has eight even-odd register pairs:

R0 – R1, R2 – R3, ... ,R14 – R15

The register pair SC may be informally described as "all 2-element register sequences that begin with an even register", and, using the definition for SB_Reg given above, can be specified as

```
SC_RegPair:  SB = SB_Reg
             Size = 2
             Restr = <0, 2, 4, 6, 8, 10, 12, 14>
```

This says that the elements of SC_RegPair are SB_Reg element sequences of length 2, whose leading member has one of the indices in the restriction sequence.

In cases where a complete integer sequence is impractical, the restriction sequence may be generated by using one of the following abbreviations. In the definitions, N is equal to or one less than the number of elements in X_{SB}:

Even	abbreviation for <0,2,4, ... ,N>
Odd	abbreviation for <1,3,5, ... ,N>
Any	abbreviation for <1,2,3, ... ,N>
j mod *k*	abbreviation for the sequence of integers such that for each integer *p*, *p* rem *k* = *j*, and *p* < N

Figure 2-6 shows the relationship between two register SCs and a register SB. Note that the relationship between SCs and SBs is many-one.

The definitions for the Vax register storage classes are shown in figure 2-7. The register storage base is defined with element size equal to 32, since registers are addressed as longwords. Note that the SCs for byte and word have elements corresponding to longwords. This is due to the fact that items of these sizes must be assigned a *minimum* of one longword register, since an instruction operand cannot directly address subfields within registers, e.g., one cannot address "the third byte in register 6."

The last four SCs represent the specific-use registers. These registers are also legal elements of the general SCs, so PACK must be careful to avoid them when binding TNs. An alternative would be to build the proper restrictions into the general SCs ("only use 0 through 11"), but other mechanisms have proved simpler in practice. Also note that when Size is larger than 1, and no specific restriction is given (Restr = ANY), then SC elements will overlap. For instance, SC_Reg_Q[0] and SC_Reg_Q[1] both use SB_Reg[1]. A set of disjoint quadwords could be defined by using the appropriate restriction:

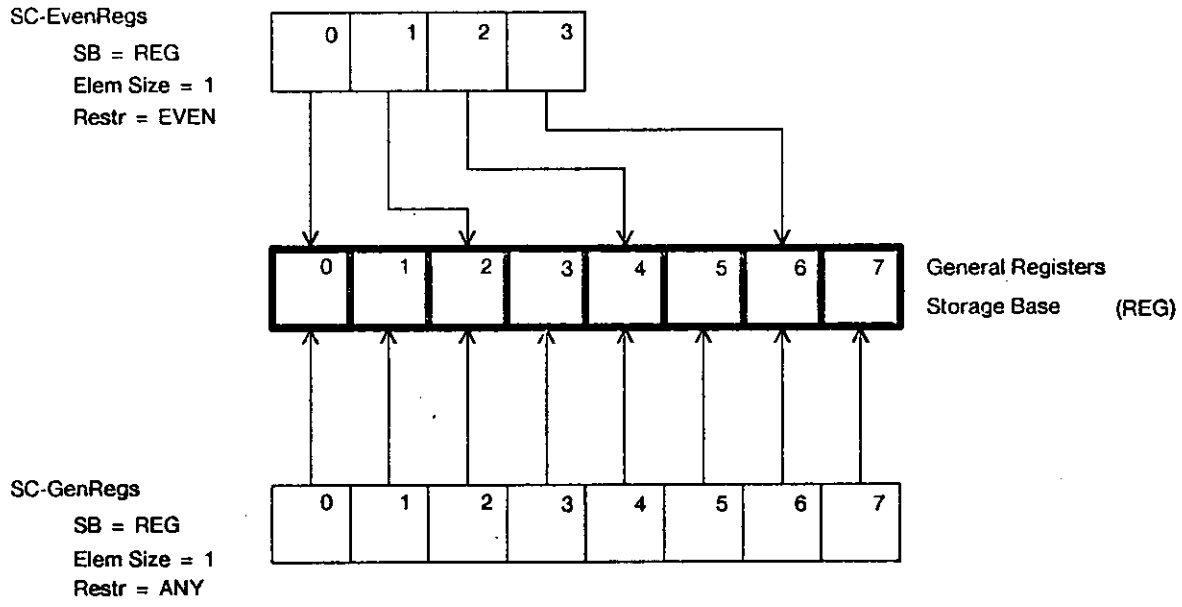


Figure 2-6: Correspondence between storage classes and storage bases

```
SC_Reg_Q: SB = SB_Reg, Size = 2, Restr = 0 mod 2
```

In contrast to the registers, Vax memory is byte-addressable. Hence, SB_Mem is defined as a sequence of byte-length storage elements. Figure 2-8 shows the SB and SCs for memory.

2.4 The Basic PACK Procedure

Packing a TN, in terms of the storage base/storage class model, consists of assigning it a SC element. This element—which, of course, corresponds to a sequence of SB elements—indicates where the variable represented by TN will reside during program execution.

The binding for a particular TN is determined in two stages. *Storage Class Selection* first selects the SC. A later step, *Location Selection*, settles on *which* element in the SC will be assigned. As will be seen, the two operations are not performed together, but at different times during the packing process.

Each TN has a *Storage Class Cost Array*, which contains information about the cost of particular binding alternatives. This structure is a sequence of cost values, indexed by SC. The values, which are based on execution speeds, are expressed in a suitable time unit[4]. For a TN *K*, the array element

[4]However, a metric based on instruction sequence size could just have easily been used (p.3).

```
Storage Base Definition
        SB_Reg: Num_Elements = 16, Element_Size = 32

Storage Class Definitions
        [Byte]          SC_Reg_B:  SB = SB_Reg, Size = 1, Restr = ANY

        [Word]          SC_Reg_W:  SB = SB_Reg, Size = 1, Restr = ANY

        [Longword]      SC_Reg_L:  SB = SB_Reg, Size = 1, Restr = ANY

        [Quadword]      SC_Reg_Q:  SB = SB_Reg, Size = 2, Restr = ANY

        [Octaword]      SC_Reg_O:  SB = SB_Reg, Size = 4, Restr = ANY

        [Prgrm Counter] SC_Reg_PC: SB = SB_Reg, Size = 1, Restr = <15>

        [Stack Pointer] SC_Reg_SP: SB = SB_Reg, Size = 1, Restr = <14>

        [Frame Pointer] SC_Reg_FP: SB = SB_Reg, Size = 1, Restr = <13>

        [Argument Ptr]  SC_Reg_AP: SB = SB_Reg, Size = 1, Restr = <12>
```

Figure 2-7: Vax register storage base and storage classes

```
Storage Base Definition
        SB_Mem: Num_Elements = 2↑32, Element_Size = 8

Storage Class Definitions
        [Byte]          SC_Mem_B:  SB = SB_Mem, Size = 1,  Restr = ANY

        [Word]          SC_Mem_W:  SB = SB_Mem, Size = 2,  Restr = ANY

        [Longword]      SC_Mem_L:  SB = SB_Mem, Size = 4,  Restr = ANY

        [Quadword]      SC_Mem_Q:  SB = SB_Mem, Size = 8,  Restr = ANY

        [Octaword]      SC_Mem_O:  SB = SB_Mem, Size = 16, Restr = ANY
```

Figure 2-8: Vax memory storage base and storage classes

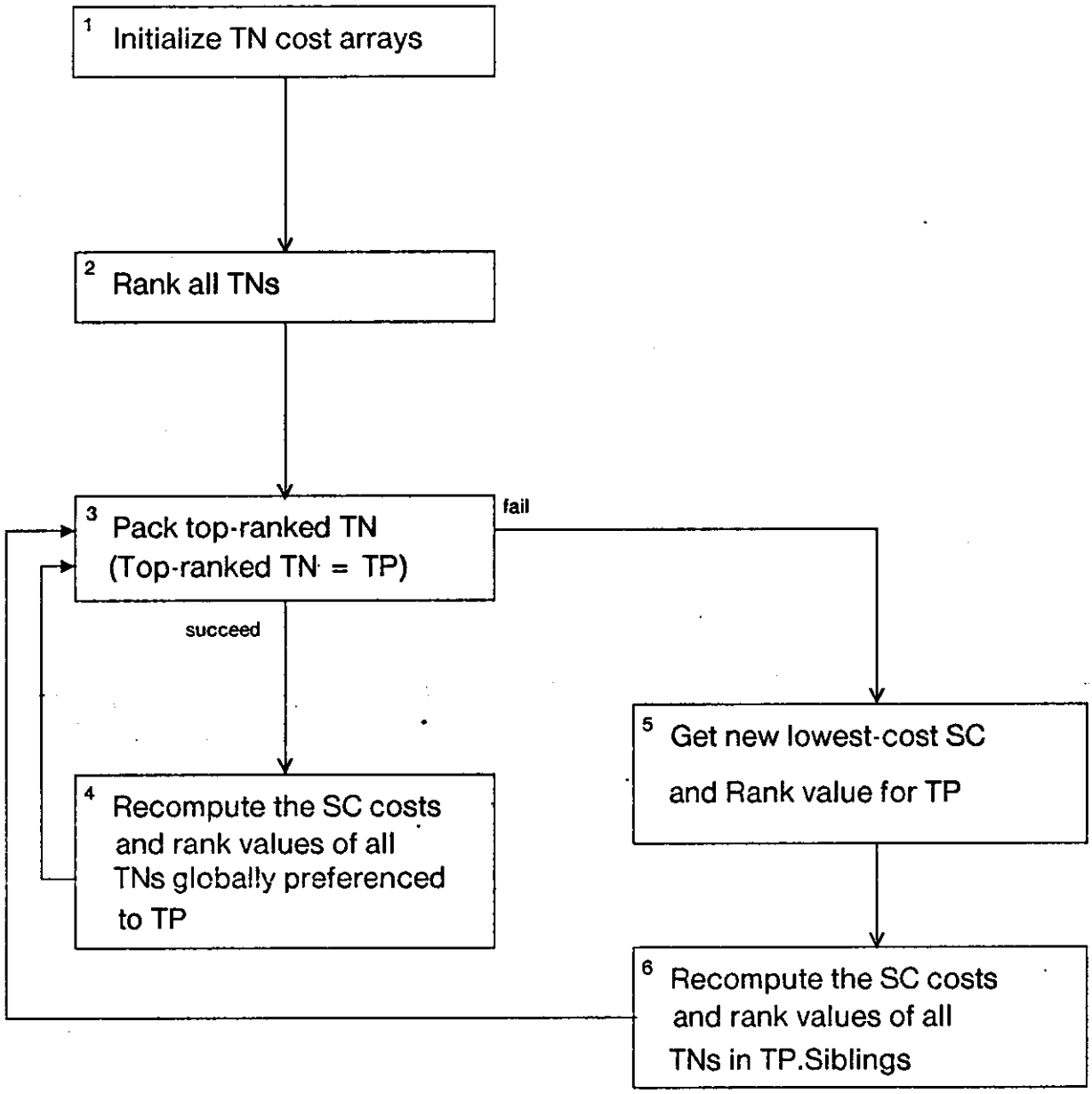


Figure 2-9: PACK Execution Flow

K.SC_CostArray[X] is an estimation of the relative cost of binding *K* to an element of SC X. The precise interpretation of these cost values will become clear when the mechanics of array construction are explained. However, all PACK needs to know is that SCs with smaller cost values are more desirable.

The basic PACK procedure is diagrammed in figure 2-9. The first step (1) initializes the SC cost arrays. When initializing, PACK makes optimistic assumptions about the quality of the code that will eventually be emitted. In particular, it is assumed that each TN will be able to reside in the SC best

suited for it—which in most cases will be some form of high-speed register. As will be seen later, when this assumption breaks down (as may happen when a TN cannot be bound to a register), the cost arrays of some other unbound TNs may be revised.

Each TN is assigned a value of importance called a *rank* in step (2). TNs with higher rank are packed first, which is desirable, because of there is less chance that a desired location will be filled by a conflicting TN. Higher rank values are given to TNs whose packing outcomes have a greater impact on program speed.

In step (3), PACK proceeds to bind the TN (called TP) with the highest rank value. Since the optimal SC for this TN has already been determined (it's the one with the lowest cost), only location selection need be performed. This is the *binding* step. In most cases involving scarce storage, properties of TNs already residing in the SC will influence which element is selected for TP.

The location selection process either succeeds or fails to find a location for TP. If it succeeds, the binding information is recorded in TP, and the TN is marked as bound. Additionally, unbound TNs that are globally preferenced to TP have their SC costs reconsidered (step (4)). In particular, the SC where TP was bound is made more favorable for these TNs, to reflect the fact that it will be profitable to eventually bind them to TP's location. The way in which this is done will be made clear when the procedure for SC cost determination is examined (section 3.3.2).

Steps (5) and (6) handle the case where location selection was unable to find an element for TP in it's favorite SC. Two things must happen: first, an alternate SC for TP is found. This is easy to do, since this SC will be the one with the next to lowest cost in TP's cost array. However, TP is not bound there immediately. Rather, a new rank value is computed, and if TP still has the highest rank value, it is bound. Otherwise, it is re-inserted in the rank list, and PACK returns to step (3).

The second thing to happen is that a certain set of unbound TNs related to TP must have their SC costs re-examined and possibly revised. This set, denoted *TP.Siblings*, contains TNs whose costs were calculated during the initialization step under the assumption that TP would be bound to it's "favorite" SC.

Steps (3), (4), (5), and (6) constitute the "PACK loop." The remainder of this chapter is devoted to these techniques.

2.5 Ranking

The construction of SC cost arrays provides PACK with an indication of the *relative importance* of the TNs. For each TN, there exists a lowest-cost SC—one where the TN should be bound in order to minimize its impact on program cost. The relative importance of the TN is gauged by the urgency with which it must be bound to this low-cost SC. This is determined by comparing the costs of the cheapest SC in the TN's cost list to the next cheapest SC. The difference is called the *rank value*:

RankValue(TN) = Cost of lowest-cost SC - Cost of next lowest SC

Consider the SC costs and rank values of two TNs that can be packed to either registers or memory:

TN_1: <Register = 2, Memory = 4>, Rank value = 2

TN_2: <Register = 2, Memory = 40>, Rank value = 38

TN_2 might appear in a loop body, where, if it is bound to memory, it would be loaded into a temporary register during each iteration. Even though the register binding cost of each is identical, it is more important that TN_2 be bound there, since program efficiency has much more to lose. In other words, if a decision must be made to pack one TN to registers and the other to memory, then TN_2 should always be given the register.

The *rank list* contains all unbound TNs, ordered by rank value. The top element on the list is the TN with the highest rank.

2.6 Storage Class Determination

Finding the cheapest SC is a matter of creating the array, and selecting the SC with the lowest value.

This task requires that PACK be able to identify a TN's role in the program. In the program tree, TNs play only one "role": Vop operands. Each TN appears in the tree two or more times—at least once as a destination operand, and at least once as a source operand. If TN_K is some TN in the tree, then TN_K's *appearance set*, denoted TN_K.App, is defined as the set of *Vop expressions* in which it occurs. A Vop expression is a tree fragment consisting of a Vop and it's operands. Figure 2-10 shows two source program uses of variable *K*, and the Vop expressions of the corresponding TN. The TN operands of a particular Vop are called *siblings*, because they all appear in the same Vop expression. TN_K.Siblings is the set of all siblings of TN_K, from all elements of TN_K.App. For example, if figure 2-10 shows the only uses of *K*, then

TN_K.Siblings = $\{TN_B, TN_C, TN_J, TN_N\}$

TN_K.SC_CostArray is constructed by gathering and combining cost data from each of its

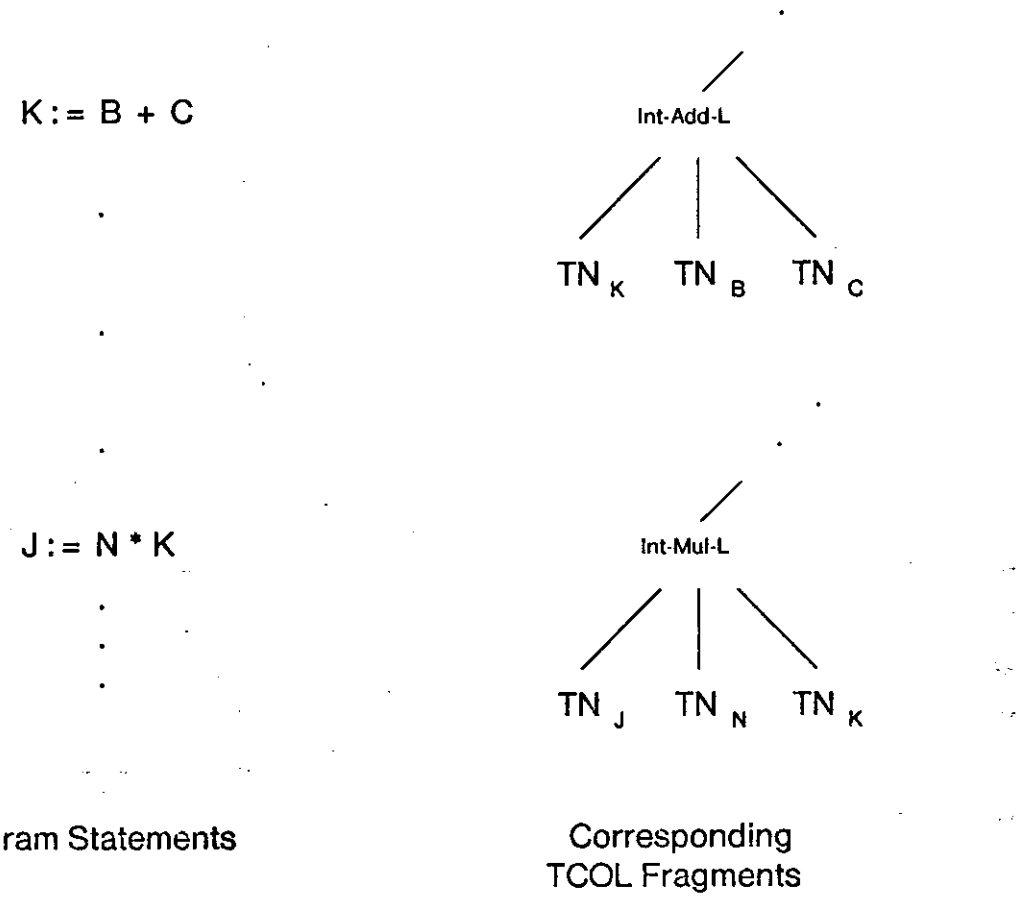


Figure 2-10: Correspondence between K and TN_K

appearances. A *Local Cost Array* LCA_i is built for each each Vop expression VX_i in TN_K.App. An LCA has the same structure as a TN cost array (cost values indexed by SC), but now the values only apply to VX_i. In particular, for some SC X, LCA_i indicates *how cheaply the Vop expression VX_i can be instantiated in the object program, given that K is bound to X*. When the all the LCAs are summed elementwise, the result is TN_K.SC_CostArray.

The storage preferences of the TN's instantiations are compounded and contrasted when the LCAs are summed. For instance, if accumulators are the desired SC in *every* LCA, then this preference will be magnified in the TNs cost array. On the other hand, if half favor accumulators over index registers by a cost difference of *J*, and half favor index registers over accumulators, also with a cost difference of *J*, the cost array will show *no* preference for either. PACK would correctly infer that from a cost-saving angle, this TN has no desire for one of these classes of storage over the other, and thus should reside in the one which is less in demand.

Generating these local cost arrays is the key to the procedure, and will be described in the next

chapter. As might be expected, properties of the machine instructions that can instantiate a Vop expression play a key role in constructing the LCA. Just as influential in the process, though, are the two sibling TNs in the Vop expression. It will be explained later how the LCA values depend on the SCs where the siblings are bound, or expect to be bound.

2.7 Location Selection

The elements of a storage class are considered to be homogeneous: no elements posess properties that would distinguish them from other elements. However, when searching for a binding for TN_K, it does not follow that all elements of a storage class X are equally suitable. TNs bound to elements of X will increase or decrease the desirability of particular locations. For example, if TN_K has a lifetime-conflict with some TN in X[e], then X[e] is out of the question as a binding. On the other hand, if if TN_K is preferenced[5] to a TN in X[e], then X[e] becomes more desirable as a binding location.

It is possible—especially when dealing with a very small register set—that each element of the selected SC X will contain a TN that conflicts with TN_K, meaning that TN_K cannot be bound anywhere in the SC. This situation corresponds to the "failure" branch of step (3) in figure 2-9. In this case, TN_K must be bound to the SC with the second smallest cost in TN_K.SC_CostArray (e.g., SC Y). However, a location in Y is <u>not</u> found immediately. Instead, a new rank value is computed, using Y as the "cheapest" SC, and TN_K is re-entered into the rank list using the new rank value.

The failure to bind TN_K has an important side-effect. It was mentioned earlier (and will be demonstrated in Chapter 3) that the creation of local cost arrays (and hence TN cost arrays) depend on knowing the "favorite" SC of sibling TNs. In other words, the TNs in TN_K.Siblings built their cost arrays under the assumption that TN_K would be bound to SC X. Since that will not be the case, and TN_K will now be bound to SC Y, the information in those cost arrays may now be incorrect, requiring their recomputation. This need only be done for members of TN_K.Siblings that haven't beem packed, since the cost arrays of bound TNs are not used further in the packing process.

[5] See section 1.1.2 for a definition of the preference relation.

3. Determining TN Storage Class Costs

The last chapter looked at the general procedure for forming SC cost arrays. The task of building an array for a particular TN can be divided into two basic steps:

1. Create a local cost array for each of the *N* tree-appearances of the TN.

2. Sum the *N* local cost arrays to arrive at the TN cost array.

This process combines local information (in the form of local cost arrays) about each appearance to find the overall cost picture. This chapter delves further into the procedure of cost array construction. In particular, nothing has been said yet about how the local cost arrays are created. This is, of course, a critical part of the procedure, and will be the first topic covered below. Other sections will cover techniques for minimizing VCC storage requirements, and the method for accounting for the impact preference relations have on SC costs.

Chapter 2 defined some terms that will be useful here. A TN_C appears in a number of *Vop expressions* as an operand. A Vop expression is simply the Vop and it's operands[6]. TN_C.App is the set of all of TN_C's appearances, one of which is illustrated in figure 3-1. TNs TN_A and TN_B are *siblings* of TN_C. The set TN_C.Siblings contains all siblings of TN_C, from all elements of TN_C.App.

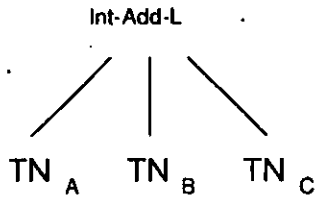


Figure 3-1: A Vop Expression *V*

3.1 Constructing Local Cost Arrays

Recall that a local cost array (LCA) has the same form as TN cost arrays: They are vectors of costs that are indexed by SC. However, the LCA will be constructed using cost information from a single Vop expression in which the TN occurs. To illustrate the procedure, let's assume that the TN cost array is being created for a TN_C, and that *V* in figure 3-1 is a member of TN_C.App. The following sections demonstrate how to build the local cost array LCA_V for *V*.

---

[6] As explained earlier (p.12), the examples used here all use three operands.

3.1.1 Deciding which SCs appear in the LCA

The target machine for this example is the Vax. The first question to ask is, which SCs will be represented in the local cost array? Figure 3-1 shows that the operator is Int_Add_L, the Vop for addition of longword integers. Observing that the Vax can handle this operation with with the operands in either static memory, registers, or stack memory, a reasonable guess for the LCA elements is

SC_Reg_L, SC_Mem_L, SC_Stk_L

There would be no reason to have an entry for SC_OddReg, an SC defining odd registers[7]. For addition, the Vax makes no distinction between an even register or an odd one. Suppose, however, that some other Vax instruction is faster if its operands reside in odd registers, and that the Vop that emits it is the operator of an element of TN_C.App. This element will have an LCA that includes the odd register SC:

SC_OddReg, SC_Reg_L, SC_Mem_L, SC_Stk_L

In this array, SC_OddReg will have a lower cost value than SC_Reg. [PACK is fully aware that these two SCs overlap, but it is "pessimistic", and will set the cost of SC_Reg according the the most expensive elements in the SC—the even registers.] In order to be able to do the summing of the LCAs properly, all must have an entry for SC_OddReg—even though it is superfluous in the context of figure 3-1. This rule can be generalized: *For a given TN, if an SC* X *appears in any of it's LCAs, then it must appear in all of them.* For simplicity's sake, assume that all the LCA's in TN_C.App contain only the three SCs for register, memory and stack. The members of TN_C.App will also be constrained to be expressions whose Vops have the same length attribute (_L) as *V*. Thus, LCA_V will have entries for SC_Reg_L, SC_Mem_L, and SC_Stk_L.

3.1.2 Filling in the Array Values

Now that the array element identities are known, the values can be calculated. In the following, TN_K.SC denotes the name of the storage class to which a TN_K is bound, and <X> denotes an element of a storage class X.

In creating LCA_V, the question being asked is, "How does the execution time (i.e., cost) of this program fragment vary as a function of TN_C.SC?" The Vop expression *V* computes an integer addition, so these costs will correspond to an instantiation of a Vax add instruction. Using the Vax **ADDL3** instruction, the costs can be found as follows, and assuming that TN_A and TN_B have been bound:

---

[7] This SC is hypothetical on the Vax, and is used for illustration only

LCA_V[SC_Reg_L] = Cost (ADDL3 <SC_Reg_L>, <TN_B.SC>, <TN_A.SC>)

LCA_V[SC_Mem_L] = Cost (ADDL3 <SC_Mem_L>, <TN_B.SC>, <TN_A.SC>)

LCA_V[SC_Stk_L] = Cost (ADDL3 <SC_Stk_L>, <TN_B.SC>, <TN_A.SC>)

If TN_A and TN_B have not yet been bound, an optimistic guess about their eventual bindings can provide values for TN_A.SC and TN_B.SC. Normally, this can be the lowest-cost SC in the unbound TNs cost array. However, if the sibling's cost array has not been initialized yet[8], then PACK can make a "best guess" about the this SC, perhaps by doing a table lookup, based on the Vop and the sibling's operand position.

This LCA has been particularly easy to construct, because **ADDL3** is the only Vax instruction that is relevant to longword integer addition, and can be used for any operand SC combination. Furthermore, the Vax also supports 3-address arithmetic, which other machines do not. A machine such as the PDP10 generally must evaluate an expression like A: = B + C in two steps:

- Perform the assignment of B or C to A (e.g., A : = B)
- Perform a 2-address addition: A : = A + C

This means that each entry in LCA_V would depend on the costs of two instructions, instead of only one, as on the Vax.

With the PDP10, there are multiple ways to code the Vop under a given SC assignment of it's operands. For instance, which is the cheaper way to begin the two-step computation of A: = B + C: Assign B to A, or assign C to A? To add to the complexity, the PDP10 has two integer addition instructions, **ADD** and **ADDM**, **ADDM** begin used when the destination TN is bound to memory.

The result is that its harder to build LCAs with the PDP10 instruction set, simply because there are multiple ways to instantiate the Vop. Since each entry of the LCA must be the *cheapest* cost, all of the alternatives must be cost-evaluated. This can be done a if a complete set of instruction timings is available, but it is not something that is desirable to do during program compilation, because of the time it consumes. Instead, the raw timing data is distilled into a tabular form during compiler construction. The resulting structures allow PACK to obtain LCA values by a simple table lookup. These are called *Vop Coding Cost* (VCC) tables.

Each Vop in the machine definition has a VCC associated with it, named VCC3_<*VopName*>. The

---

[8] A situation that might arise during PACK's initialization step (p.18).

table is three-dimensional, and is indexed by storage class. Each table entry is a cost, and indicates the execution time of the lowest-cost instantiation of a Vop expression, given the indicated set of SC assignments for the three operands.

For example, return to figure 3-1. The value for the local cost array entry LCA_V(SC_Reg_L) is contained in the following entry of VCC3_Int_Add_L:

VCC3_Int_Add_L(*A*.SC,*B*.SC,SC_Reg_L)

Similarly, the values for the remaining entries of LCA_V can be obtained from this table.

Any local cost array can be built in this way with the help of a VCC table. The building of the tables themselves is the subject of the next section.

3.2 Building and Managing the Vop Coding Cost Tables

As seen above, each Vop has a VCC table associated with it. However, because of a special condition that commonly shows up in Vop expressions, each Vop needs a smaller, supplementary VCC table, in addition to the ones described earlier. This condition occurs when one of the source operands also serves as the destination operand. Such a Vop expression is called a two-operand expression (fig. 3-2).

Figure 3-2: (a) A three-operand Vop expression. (b) A two-operand Vop expression

Two-operand Vop expressions are generally cheaper to code. As a result, the entries in the three-operand VCCs discussed earlier aren't valid for them, hence the need for supplementary tables. These tables are prefixed by "VCC2_", to distinguish them from the three-operand tables, and are two-dimensional, not three-dimensional. Otherwise, they work just like the larger ones. For instance, the the cost of the expression in fig. 3-2(b) is in the table element

VCC2_Int_Add_L (TN_A.SC, TN_B.SC)

A pair of tables must therefore be built for each Vop. The same basic procedure is used to create

both types of tables, and will be illustrated below by building tables for the integer addition Vop. To show the generality of the VCC table technique, tables will be built for both the Vax and the PDP10. Later on, the issue of table compaction will be examined. It will be shown that the total storage needed for a full set of tables might be large, and that some simple techniques can be used to shrink VCC tables, or share a single table among multiple Vops.

3.2.1 Constructing VCC Tables

On the Vax, register-length integers are stored in *longwords*, and on the PDP10, they are stored in *words*. Accordingly, the Vop representing VAX addition of these quantities is Int_Add_L, and Vop for PDP10 addition is Int_Add_W. These two Vops will be used to illustrate the VCC table construction process.

The Vax has an instruction set that includes two-address forms of nearly all arithmetic instructions. Like addition, most instructions may have their operands in any SC. Thus, the basic two-address add instruction can be used to define all entries in the two-address VCC:

```
For all SCs A and B,

  VCC2_Int_Add_L(A,B) = Cost of ADDL2 <A>, <B>
                        Where <X> = an element of SC X
```

A and B range over the set {SC_Reg_L, SC_Mem_L, SC_Stk_L}. Thus, all of the elements of VCC2_Int_Add_L correspond to the cost of an instantiation of **ADDL2** with different combinations of operand assignments.

The construction procedure is nearly the same for the PDP10's VCC2_Int_Add_W, except for the case where both operands are in memory. No simple instruction can handle it: there must be an additional instruction to load one of the operands into a temporary register. If a and b are in SC_Mem_W, then a := a + b is coded as

```
MOVE TempReg, b
ADDM TempReg, a
```

and the table entry is computed accordingly:

```
VCC2_Int_Add_W(SC_Mem_W,SC_Mem_W) = (Cost of MOV b,TempReg)
                                  + (Cost of ADDM TempReg, a)
```

All the other elements of the table are the costs of single instructions—either **ADD** or **ADDM**. Table 3-1 shows the entry values for the PDP-10's VCC_Int_Add_W, and the code fragments that generated them. All timings in this table are for the PDP10 KA10 implementation ([DEC 69]). Note also that there is no stack SC on the PDP10.

Table element	Value	Code Sequence
(SC_Reg_W, SC_Reg_W)	2.19	ADD R1, R2
(SC_Reg_W, SC_Mem_W)	3.08	ADDM R1, M
(SC_Mem_W, SC_Reg_W)	2.53	ADD R1, M
(SC_Mem_W, SC_Mem_W)	5.29	MOVE Rt, M2 ADDM Rt, M1

Rt is a temporary register to be allocated by CODE

Table 3-1: VCC2_Int_Add_W for the PDP-10. Values are in microseconds.

Table element	Value	Code Sequence
(Reg, Reg, Reg)	4.06	MOVE R1, R3 ADD R1, R2
(Reg, Reg, Mem)	4.40	MOVE R1, R2 ADD R1, M3
(Reg, Mem, Reg)	4.40	MOVE R1, R3 ADD R1, M2
(Reg, Mem, Mem)	4.74	MOVE R1, M3 ADD R1, M2
(Mem, Reg, Reg)	5.55	MOVEM R3, M1 ADDM R2, M1

(Mem, Reg, Mem)	6.87	MOVE R2, Rt
		ADD M3, Rt
		MOVEM Rt, M1

(Mem, Mem, Reg)	6.87	MOVE R3, Rt
		ADD M2, Rt
		MOVEM Rt, M1

(Mem, Mem, Mem)	7.21	MOVE M3, Rt
		ADD M2, Rt
		MOVEM Rt, M1

Reg = SC_Reg_W Mem = SC_Mem_W

Rt is a temporary register to be allocated by CODE

Table 3-2: VCC3_Int_Add_W for the PDP-10. Values are in microseconds

It is clear that these tables have a lowest-cost element, or set of elements, corresponding to the most desirable operand assignment(s). The other assignments represented in the table incur excess cost either because they require relatively expensive instructions, or because they require operand loading instructions, or both.

Consider now the three-address tables. The Vax has three-address instructions that make the table construction as straightforward as the two-operand case. Not so with the PDP-10. Since it only supports two-address instructions, one of the source operands must be loaded into the destination location before the arithmetic computation. Table 3-2 shows the values and code sequences.

The general rule for computing table entries is to always try to find a single instruction that can handle the operation under the given operand binding constraints. If that fails, then some other instruction must be used along with a load or store. Finding the cheapest combination may involve some search.

3.2.2 Minimizing VCC Table Size

It is easy to see that the size of the VCC tables is related to the number of SCs that can serve as binding sites for the Vop's operands. For instance, each operand of the Vax's Int_Add_L can potentially be bound to any one of the set {SC_Reg_L, SC_Mem_L, SC_Stk_L}. Since there are three possibilities for each of three operands, the number of elements in VCC3_Int_Add_L is $3^3 = 27$. In general, the number of elements in a three-operand VCC table equals N^3, where N is the number of SCs whose element length is equal to the length attribute of the Vop (_L, in this case). Since there is no fixed upper bound on the number of SCs in a machine description, or the number Vops, they might take up a considerable amount of space. However, there are some methods for minimizing the total size of the tables.

One way of managing the space requirement is to combine the tables of Vops that differ only in their data length. The Vax's Int_Add_B, Int_Add_W, and Int_Add_L are three such VopS

The idea is to create one table for the "cheapest" Vop—Int_Add_B, in this case. Table values are derived for the other Vops by adding in the extra cost of using longer operands, and computing a longer result. Thus, the table entry in VCC3_Int_Add_W for the three-address expression

```
a := b + c        [a,b in registers, c in memory]
```

may be found by using the corresponding entry in VCC3_Int_Add_B:

```
VCC3_Int_Add_W(Reg,Reg,Mem) = VCC3_Int_ADD_B(Reg,Reg,Mem)
                                + ExcessOpCost_Word(Reg,Reg,Mem)
                                + ExcessCompCost_Word(Int_Add)
```

ExcessOpCost_Word is a function that finds how much more expensive it is to read the two source operands, and write the destination operand, if they are word-length instead of byte-length. ExcessCompCost_Word returns the additional cost of computing a word-length result over a byte-length result. Two similar functions exist for Longwords.

This compaction technique can be used whenever the operand access costs and the result computation costs can be easily partitioned, as they can on the Vax.

A second, more general method, shares rows or columns in the table among storage classes that have the same cost characteristic with respect to the Vop. A general register machine may, for instance, allow the operands for an addition to be in any of the registers. For such a machine, it does not make sense to have a separate row and column for specialized register SCs, such as EVEN. Since the costs of operands in an even register are clearly going to be equivalent to those for the general register SC, one row and column may be used for both SCs. Figure 3-3a shows a VCC table with separate rows and columns for the register and even register SCs. Figure 3-3b shows how the table may be compacted by sharing.

The last compaction technique takes advantage of the similarities that are bound to arise in tables for similar arithmetic Vops. If two such Vops, such as addition and subtraction, give rise to tables that differ by a scalar quantity, then they can both share a single table. For instance, if

VCC_A = VCC_B * N + K

where VCC_A and VCC_B are tables of identical dimensions, and N and K are scalars, then any element of VCC_A can be derived by multiplying the corresponding element of VCC_B by N and adding K to the result. It is possible that more than two Vops may be related in this way, and share one table among them.

3.3 The Preference Relation

3.3.1 Local Preferencing

The coding costs of three address fragments that were developed above are a bit too pessimistic. If PACK manages to bind one of the source operands to the the same location as the destination operand, then the CODE can treat the fragment like a two-address case (figure 3-4). Looking back at our examples above, this optimization saves execution time difference between a two-operand and three-operand instruction on the Vax, and saves a move instruction on the PDP-10.

	SC-Reg-W	SC-EvenReg-W	SC-Mem-W
SC-Reg-W	2.19	2.19	3.08
SC-EvenReg-W	2.19	2.19	3.08
SC-Mem-W	2.53	2.53	5.29

(a)

	SC-Reg-W SC-EvenReg-W	SC-Mem-W
SC-Reg-W SC-EvenReg-W	2.19	3.08
SC-Mem-W	2.53	5.29

(b)

Figure 3-3: Two equivalent VCC tables

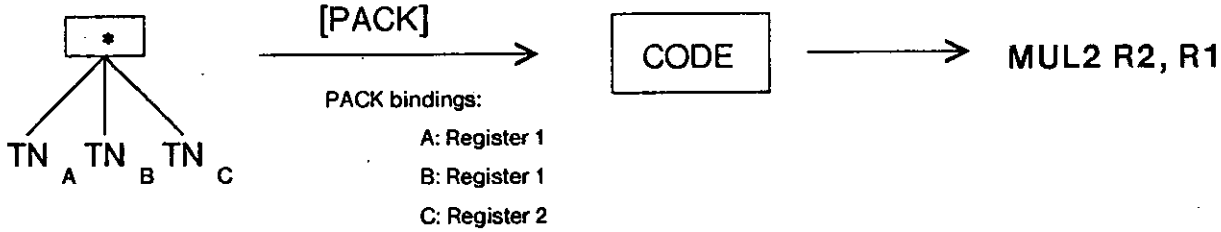


Figure 3-4: Packing a source operand with the destination allows a three-address fragment to be coded more efficiently

The desirability of binding two TNs to the same machine location is transmitted to PACK by

establishing a *preference link* (or preference relation) between the two TNs. The relation described in the previous paragraph is called a *local* preference link. Global preference links will be described later. In the expression representing A : = B + C, local preference links exist between A & B, and A & C.

Satisfying local preference relations—packing the two TNs to the same location—is ultimately the responsibility of the location selection procedure. Location selection may not be able to honor a preference link, or it may choose not to.

It is also worth investigating what relationship the local preferences have to the SC determination procedures, to see in what manner a TN's local preferences affect the relative attractiveness of the storage classes.

Unfortunately, it is very difficult to make quantitative judgments about local preference during SC determination. The problems arise because the excess cost—e.g., the need to use a 3-op instruction—can be avoided by two independent TN binding decisions. Taking the expression

A : = B + C

as an example, the extra cost is saved if A and B end up together, *or* A and C end up together. It is hard to express the necessary cost information accurately in a single TN's cost record. A more global recordkeeping system would be needed.

Even if the problem can't be solved quantitatively—by assigning extra costs to one or more SCs in a TN's cost list—there is still a temptation to do something to "boost" a local-preferenced TN in the rank list after the other TN in the preference link has been packed to a scarce location. This is a questionable optimization, because the distance the TN is raised in the rank list will be arbitrary by nature, and will only become reasonable by "tuning". It is also worth remembering that such an optimization attempt is bound to be cancelled out in large part because *every* TN associated with a three-operand instruction will be preferentially treated in the same way.

3.3.2 The Global Preference Relation

Situations arise where it is desirable for two TNs to occupy the same location in storage. For instance, when

A := if *e* then (B+C) else (B-C)

is coded, the destination TN for both expressions (B + C) and (B – C) should be TN_A. An earlier phase performs *targeting*, which determines the feasibility and desirability of of doing this. For certain reasons, this phase may be unwilling or unable to use TN_A as the destination for both subexpressions.

When this happens, a temporary TN is created for the untargeted subexpression, in lieu of TN_A—for instance, $TN_{(B+C)}$. The object program will contain an instruction to move the contents of $TN_{(B+C)}$ to TN_A, *unless* the two TNs are bound to the same location. PACK is left with the responsibility of binding the two to the same storage location.

Two TNs like TN_A and $TN_{(B+C)}$ are related by the *global preference relation*. The Bliss/11 compiler proved the utility of using them in storage assignment [Johnsson 75, Wulf 75]. A global preference (hereafter, simply "preference") link tells PACK that the two TNs ought to be bound together in storage. If they are not, then an extra move instruction will be needed to move one into the location of the other. The excess cost of not heeding the link is extra time needed for this transfer.

Both SC cost determination and location selection have a part in the management of preference costs. When the first of two preference-related TNs is packed, its binding affects the SC costs of the second, unpacked TN. And if that other TN is finally packed in the first one's SC, the location selection process is responsible for correctly weighing the desirability of packing the two together against other, possibly conflicting costs. The location selection procedure will be described later on.

The first step in understanding how PACK manages preference costs is to observe that one of the TNs can be characterized as the "source" of the potential data move, and the other can be labeled the "destination". In the example above, the temporary $TN_{(B+C)}$ is the source, and TN_A is the destination. The cost that PACK tries to avoid is a data transfer between the source and the destination.

These costs are worked into the TN storage class cost arrays in the manner shown in the following example. Suppose TN_s and TN_d are preference-related, with the source being TN_s and the destination being TN_d. Now there is little we can say about the preference link cost until one of the TNs is bound by PACK. Suppose TN_s is packed first, to some SC, say TN_s.SC. How does this affect the cost of binding or not binding TN_d to a location in TN_s.SC? If TN_d is not packed in TN_s.SC, then a data move between a location in TN_s.SC and a location in this other SC, TN_d.SC will certainly be needed. Thus, the relative cost of binding TN_d to any SC other than TN_s.SC is increased by the cost of this data transfer. On the other hand, if TN_d *is* packed into TN_s.SC, then at the very most, all that will be needed is an intra-SC move between one location of TN_s.SC and another. If things work out, of course, both TNs will be in the same location and no move will be necessary, but it pays to be conservative at this point and assume the worst case. This reasoning is reflected in the cost setup by adjusting N_d.SC_CostArray shortly after TN_s is packed:

```
For each SC X in TN_d.SC_CostArray,

    TN_d.SC_CostArray[X] := TN_d.SC_CostArray[X]
                            + K * SC_TransCost(TN_s.SC, X)
```

SC_TransCost is a two-dimensional table supplied in the machine description that indicates the cheapest cost of doing a data transfer between elements of two storage classes. SC_TransCost(X,X) is the cost of doing a transfer between one element of an SC X and another. The value *K* is an "importance" factor associated with the preference link, i.e., loop depth.

If TN_d had been packed before TN_s, then a similar computation would be performed on the SC cost array of TN_d:

```
For each SC X in TN_s.SC_CostArray,

    TN_s.SC_CostArray[X] := TN_s.SC_CostArray[X]
                            + K * SC_TransCost(X, TN_d.SC)
```

The difference is that the order of the indices in SC_TransCost have been switched.

The global preference system helps steer PACK in the right direction when it is confronted with preference requests. It is important to know *how* cost-saving a preference link is, in addition to knowing that the link exists. The system above makes a storage class level estimation of this information, and integrates it into the SC cost arrays.

4. Location Selection

Location selection is the PACK subphase that that performs the final binding of a TN. SC cost determination has found the most desirable SC, and now location selection searches that SC for the "best" element.

Leverett [Leverett 81] developed a method called *voting* for finding the favored location. Each SC element is examined in turn, and its desirability rated according to a set of criteria. When all elements have been scored, the least-cost one is selected. The criteria for voting that are explained below are based loosely on Leverett's originals.

Recall that an SC element is really an element or sequence of elements in its root SB. We saw earlier that many SCs might have the same root SB, e.g., the Vax longword and quadword register SCs. When scoring an SC element, PACK is interested in what TNs are bound there. PACK must realize that a TN bound to SC_Reg_Q[0] (registers 0-1) is also "bound" to SC_Reg_L[0] (register 0), because the definitions of these elements overlap. Elements of different SCs with the same root SB interact according to the following two rules. Suppose A and B are two SCs with the same root SB. *a* is an element of A, and *b* is an element of B:

1. A TN bound to *a* and a TN bound to *b* conflict if *a* and *b* share any SB element, as in the longword/quadword example above.

2. *a* and *b* are *equivalent* if the first SB element of both are the same. This definition is useful for preferencing TNs whose SC elements differ in size. For example, SC_Reg_O 1 (registers 4-7), SC_Reg_Q 2 (registers 4-5), and SC_Reg_L 4 (register 4) are all equivalent.

A few abbreviations will help to explain the voting procedure:

- TN_p is the TN undergoing packing—the one that is searching for a location.
- $TN_p.Conflicts$ is the set of TNs which have lifetime conflicts with TN_p.
- $TN_p.Pref$ is the set of TNs that are (locally or globally) preference related to $TN_p.Pref$

In the voting definitions below, the SC element under consideration is *L*, and its current voting score is *L*.score.

L is scored as follows:

1. If a member of TN_p.Conflicts is in *L*, then add an infinite cost to *L*.score, since it cannot be used.

2. If L contains a member of TN_p.Pref, then *subtract* the cost of the data move saved by the preference link from L.score.

3. If there exists an unbound member of TN_p.Pref—call it TN_u—such that a member of TN_u.Conflicts is bound to L, then it will not be possible to honor the preference link between TN_p and TN_u if TN_p is bound to L. L.score is increased by the cost of the data move between TN_p and TN_u.

4. If there exists some TN which

   - is unbound
   - is a member of TN_p.Conflicts
   - is preferenced to a TN in L

   then packing TN_p to L will eliminate the chance of honoring the preference link. It is unclear what should be added to L.score. It is not proper to simply add the cost of not honoring the preference, because there might be other reasons why the link is not honored. In other words, the "fault" for killing this preference opportunity may not devolve completely on TN_p.

The location with the lowest score is selected. If all of the locations have an infinite cost, then this is an indication that a member of TN_p.Conflicts is present in every location in the SC. TN_p will not be able to be packed in the SC, and must settle for the next most cost-effective one.

The voting method does an exhaustive search among all locations of the SC for the lowest-cost one. This is impractical if the SC has a large number of elements, as will those SCs representing memory. An implementation of the location selection scheme will have to be able to do a limited search. This can be done by working backwards from the information in TN_p.Conflicts and TN_p.Pref, to first scoring locations that are promising. For instance, the voter might first score all locations that contain a member of TN_p.Pref.

5. Evaluation

5.1 Performance

This PACK design has not been implemented, but the similarities between it and an earlier PQCC PACK phase would indicate that the quality of its packing should be equal to or better than the earlier design.

The old and new are alike in enough ways to make the comparison reasonable. Both have the same role in the PQCC phase structure, and add the same type of information to the program tree—namely, the TN binding information. The general layout of both designs are similar. Like the design presented here, the earlier PACK has costs expressed as SC-indexed cost arrays associated with TNs. TNs were selected for packing on the basis of relative importance, as they are here.

The major difference between the designs is the quality of the cost information. The earlier design didn't make direct comparisons between the execution-time cost consequences of competing binding alternatives. The costs contained in the SC arrays were more abstract, indicating the relative expense of SC use. The value for registers might be one "point", and the value for memory might be two "points". In this case, registers are clearly cheaper to use than memory, but the magnitude of desirability was vague.

The new design has more resolution in the cost information, because it uses instruction timing information such as that provided in the VCCs. This style of cost determination also allows the global preference information to be used much more effectively. However, as pointed out above, the basic philosophy of the two designs are similar, so the change (for the better) in information quality should result in better packings.

5.2 Retargetability

The discussion on retargetability in the Introduction pointed out that in a retargetable compiler, the machine dependent information should be clearly identifiable, and easily changable. PACK's machine description interface was designed to be acceptable under these criteria.

PACK depends on two types of machine information. First, there is the knowledge about storage layout, as represented by the storage classes and storage bases. The second type is the machine instruction timing information from which the Vop coding cost tables are derived.

Experience has shown that the storage base and storage class notations are sufficient to describe a wide range of general register architectures. The techniques of section 2.3 show how a set of storage classes and stoage bases can be defined for these machines.

The instruction information needed is not complex. PACK is unconcerned with the semantic properties of instructions, but rather only needs to know how execution speeds change as a function of operand location. This is a well-defined knowledge requirement, and it (the knowlege requirement, not the knowledge) remains constant from machine to machine.

The issue with this type of knowledge is not its complexity, but the amount needed. It was shown earlier that the combined size of the Vop coding cost tables might be considerable. Hand-building them would be a long and error-prone process, since calculating most table entries invovles some search amongst candidate instructions or sequences. However, the construction task can be automated. Instead of supplying Vop coding cost tables for each Vop, the retargeter would only supply cost information about each machine instruction. The VCCs would be built by the compiler-writing system during compiler retargeting.

The information provided by the retargeter for each instruction is similar in nature to the information that eventually appears in VCCs_namely a compilation of execution times as a function of operand location. Figure 5-1 shows this data for some PDP10 arithmetic instructions.

Operand Locations: ***Src, Dst***

	Reg, Reg	Mem, Reg	Reg, Mem
ADD	2.19	2.53	3.08
SUB	2.19	2.53	3.08
IMUL	9.25	9.59	10.56
IDIV	16.2	16.5	17.4

Figure 5-1: PDP-10 Instruction cost information, in microseconds.

The only remaining instruction cost data needed are the values of the SC_TransCost array used for global preference processing (section 3.3.2):

SC_TransCost(A, B) = cost of a data move between a location in SC A and a location in SC B.

SC_TransCost(A, A) = cost of a data move between one element of A and another.

All of the data structures described here share a very important property: The person building them does not have to know how PACK or the rest of the compiler works. The only requirement is a familiarity with the machine.

Acknowledgements

The style and content of this paper benefited from the suggestions of my advisors on this project: Bill Wulf, Joe Newcomer, and Kesav Nori. Bruce Leverett also provided valuable comments on an earlier draft.

References

[Brosgol 80] B. M. Brosgol, J.M. Newcomer, D.A. Lamb, D. Levine, M. S. Van Deusen, and W.A. Wulf.
$TCOL_{Ada}$: Revised Report on An Intermediate Representation for the Preliminary Ada Language.
Technical Report CMU-CS-80-105, Carnegie-Mellon University, Computer Science Department, February, 1980.

[DEC 69] *PDP-10 Reference Handbook*
Digital Equipment Corporation, 1969.

[Fabri 79] Janet Fabri.
Automatic Storage Optimization.
In *SIGPLAN Conference on Compiler Construction*, pages 81-91. ACM, Denver, 1979.
Published in SIGPLAN Notices, Aug 1979.

[Goos 81] G. Goos and W. A. Wulf (editors).
Diana Reference Manual.
Technical Report CMU-CS-81-101, Carnegie-Mellon University, Computer Science Department, March, 1981.

[Johnsson 75] R.K. Johnsson.
An Approach to Global Register Allocation.
PhD thesis, Carnegie-Mellon University, December, 1975.

[Leverett 82] B. W. Leverett.
Topics in Code Generation and Register Allocation.
Technical Report CMU-CS-82-130, Carnegie-Mellon University, Computer Science Department, july, 1982.

[Leverett 83] Bruce W. Leverett.
Register Allocation in Optimizing Compilers.
UMI Research Press, 1983.

[Leverett 80] B.W. Leverett, R.G.G. Cattell, S.O. Hobbs, J.M. Newcomer, A.H. Reiner, B.R. Schatz, W.A. Wulf.
An Overview of the Production Quality Compiler-Compiler Project.
Computer 13(8):38-49, August, 1980.

[Leverett 81] B.W. Leverett.
Register Allocation in Optimizing Compilers.
PhD thesis, Carnegie-Mellon University, February, 1981.

[Wulf 75] W. Wulf, R.K. Johnsson, C.B. Weinstock, S.O. Hobbs, and C.M. Geschke.
The Design of an Optimizing Compiler.
American-Elsevier, 1975.

[Wulf 80] W.A. Wulf.
PQCC: A Machine-Relative Compiler Technology.
In *IEEE 4th International COMPSAC Conference*, pages 24-36. Chicago, October, 1980.
Also available as CMU Technical Report CMU-CS-80-144.