CMU-CS-82-130

Topics in Code Generation and Register Allocation

Bruce W. Leverett

Carnegie-Mellon University
Computer Science Department
Pittsburgh, Pennsylvania 15213

Abstract

This paper discusses some questions about register allocation and code generation in optimizing compilers. The context of the research is the PQCC (Production Quality Compiler-Compiler) project. The questions discussed include fundamental questions of compiler structure, that is, questions of the feasibility and correctness of the approach taken by the project. I also report on less fundamental issues, issues more or less orthogonal to the questions of structure. This discussion should be of interest to any designer of optimizing compilers who is interested in *retargetability*, that is, in the adaptability of such a compiler to modification to allow code generation for different target machines.

28 July 1982

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

Introduction

This paper discusses some questions about register allocation and code generation in optimizing compilers. The context of the research is the PQCC (Production Quality Compiler-Compiler) project. The questions discussed include fundamental questions of compiler structure, that is, questions of the feasibility and correctness of the approach taken by the project. I will also report on less fundamental issues, issues more or less orthogonal to the questions of structure. This discussion should be of interest to any designer of optimizing compilers who is interested in *retargetability*, that is, in the adaptability of such a compiler to modification to allow code generation for different target machines.

Overviews of the PQCC project have been given in other papers.[8, 9, 14] The following statement of its research goals is taken verbatim from one of those papers:[8]

> The Production Quality Compiler-Compiler (PQCC) project is an investigation of the code generation process. The practical goal of the project is to build a truly automatic compiler-writing system; compilers built with this system will be competitive in every respect with the best hand-generated compilers of today. They must generate highly optimized object code, and meet high standards of reliability and reasonable standards of performance. The system must operate from descriptions of both the source language and the target computer. The cost of bringing up a new compiler, given a suitable language description and target architecture description, must be small — on the order of three man-months, without assistance from builders, maintainers, or other persons deeply involved in the original system.

After living out the usual life-span of large academic research projects, the PQCC project was reduced substantially in size and level of activity in July of 1981. This followed the demonstration of a system, various parts of which fit various parts of the above description to varying degrees. Now, therefore, seems as good a time as any for a summary of the results of the project.

The contribution of this paper to that goal is primarily as a postscript to a comprehensive treatise by the same author, *Register Allocation in Optimizing Compilers*[10] (hereafter referred to as RAOC). We have learned a great deal of a practical nature since that was published in February of 1981, particularly in the weeks immediately preceding the demonstration in July. This paper is meant to be self-contained, that is, the reader should not have to read the larger work in order to understand this one. It should be emphasized, however, that this paper is not an overview of the project, but an updating of research results and conclusions.

1. Code Generation and Code Generator Generation

We designated a set of phases, operating serially, into which any PQCC-generated compiler would be divided. Associated with each phase was a *phase generator* or *phase generation program*, which, given straightforward descriptions of the target machine and/or the language to be compiled, was to extract and analyze information necessary for the operation of the phase, and to organize the information into tables

suitable for direct use by the phase. (Thus the phase generator runs at "compiler-compilation time," and the phase itself runs at "compilation time.") I use the word "tables" in a very broad sense; some of the phase generators produced code, which could be linked in with the compiler. In most cases, however, a phase consisted of code that was independent of the target machine and language (and therefore could be common to all generated compilers) and tables holding machine- and language-dependent information.

Below I will use the term *code generation* to refer to a series of phases, and *code generator generator* (*CGG*) to refer to the set of phase generators for those phases. The phases we include under this heading perform both *register allocation*, the construction of a mapping between data values (and objects) and target-machine storage locations, and *code selection*, the choice between alternative target-machine code sequences to implement each source-program action. In the PQCC organization of phases, there are two phases that perform code selection, as described in Section 2. Each uses a library of *code selection templates*. A template consists of a *pattern*, a description of some source-language feature or combination of features, and a *code sequence* that could be used to implement that feature. Typical features are arithmetic and logical operators; for example, a template might describe an implementation of single-precision integer addition. If a target machine has several instructions each of which could implement this operator, each would be represented in the library by its own template, and the different templates would have very similar patterns. As the source program is represented during compilation by a tree, the patterns likewise take the form of trees. We use the term *pattern matching* for the action of a code selection phase. Starting at the root of the source-program tree, it finds applicable templates for each operation, choosing one before proceeding downward by the tree branches.

We reconsidered the division of labor among the compiler, the phase generation programs, and the person who draws up the target machine description. The critical question for the practical compiler builder is about the extent to which he should develop or refine his tools, e.g. the phase generators, given that his goals may be more modest and his constraints more severe than those of the PQCC research project. Section 1.1 discusses this question and describes our own response to considerations of practicality, as reflected in the design of the CGG. Another important question, more or less orthogonal to the first, concerns the efficiency of code selection: are there obstacles that prevent a pattern-matching code generator, such as we used, from competing in efficiency with more conventionally hand-coded systems? Section 1.2 summarizes some of the sources of inefficiency in the pattern matchers we implemented, and explains how these can be dealt with.

1.1. Development of Tools

The CGG proposed, and implemented in prototype, by Cattell[2] performed *search*. Thus, it found non-trivial connections between source-language constructs and target-machine features. In addition, it performed the low-level function of *assembly*: the code generation templates were sorted and classified; information that had been human-readable in the original machine description was compressed to a format suitable for direct use by the compiler; cross-references between different parts of the code generation tables were recorded. Search and assembly are obvious candidates for automated processing in a CGG; in addition, *analysis* of the machine description to derive various summary data, or even to construct large tables, is required for most phases.*

Because more than one phase, in the PQCC design, requires information derived from the search process, we were required to separate search from assembly. To do this, and for other reasons, we abandoned Cattell's prototype CGG and set about to construct separate search and assembly programs from scratch. (Analysis was to be done by the search program, although in principle it might best be done by the assembly programs for the separate phases, since each phase has its own analysis requirements.) The input to any assembly program, and the output of the search program, was to be in an intermediate format, human-readable but including various rather cryptic notations giving the results of analysis. At this point, however, "considerations of practicality" were allowed to intervene decisively. We never rebuilt the search and analysis program; instead, in about one man-week of effort by personnel who were thoroughly familiar with the PQCC design, we drew up a single machine description in the intermediate format. Later, we constructed other machine descriptions, but only the first one was tested and debugged. (Appendix II gives some representative pages from that description, which was for the DEC VAX-11.)

This compromise was motivated, not simply by project deadlines, but also by the prospect that the system would be tested only for a small number of target machines. The project goal of using machine descriptions that could be drawn up by persons unfamiliar with compiler technology, or at least unfamiliar with PQCC technology, was not achieved. With regard to the equally important goal of using the same machine description to provide tables for several phases, what we achieved was quite satisfactory: almost all the tables and all the template libraries used by the code generation phases were assembled from the same intermediate-format machine description. I am convinced that the effort spent to attain this end saved us countless headaches in the compiler implementation and debugging.

---

*The preliminary function of *symbolic execution*, or derivation of a machine description in Cattell's MOP format from a procedural format such as ISP,[1] was implemented in prototype by Oakley[11] but was never incorporated into the PQCC system.

1.2. Efficiency of the Pattern Matcher

In the total system that we had implemented as of July 1981, the code selection phases were not bottlenecks (see Appendix I). Nevertheless, some obvious sources of inefficiency in the design of our pattern matchers have been brought to our attention. I will discuss them here, as they could be important in the construction of systems in which the speed of the code selector is critical to the speed of the compiler as a whole.

Consider the task of selecting between equivalent instructions or sequences to implement an operation, sequences differing only in the storage requirements they impose on their operands or in the classes of storage in which they leave their results. An example of this selection, also used in RAOC,[10, ch. 11] is taken from the IBM S/370. The logical AND operation between four-byte operands on this machine is done with one of three instructions, whose mnemonics are NR, N, and NC. Denote the *target* operand (the operand whose contents will be replaced by the result of the operation) as **t**, and the other (non-target) as **nt**. The choice among the three instructions depends on the storage allocated for **t** and **nt**, as shown in Figure 1-1:

Storage classes used		Best code sequence		
T	NT			
accumulator	accumulator	NR	T, NT	
accumulator	memory	N	T, NT	
memory	memory	NC	T(4), NT	
memory	accumulator	ST	NT, X	; store **nt** to memory
		NC	T(4), X	

Figure 1-1: Implementations of **t** := **t** & **nt** on the IBM 370

Note that the last code sequence requires a free memory location, denoted **x**; since **nt** cannot be used by the NC instruction in this case, its contents are copied into **x**, and **x** is used.

This example illustrates several features that are characteristic of such code selection problems. First, in one of the cases (the last one), one of the operands will have to be copied to a free (temporary) location no matter which of the three instructions is used; it happens here that the best code sequence is one in which **nt** is copied. I refer to this copying as *operand loading*. Second, the best code sequence for any case is not the only code sequence for that case; in fact, with suitable operand loading (and possibly restoring of the result), any of the three instructions could be used for any of the four cases. For instance, the NR instruction could be used for the second case, if **nt** is first copied into a free accumulator.

In the main code selector of our July 1981 implementation, the choice of the best code sequence for each case is done by comparison of code sequence and operand loading costs recorded in the machine description. Thus, each of the three instructions (N, NR, NC) has a *base cost* (perhaps its size in bytes, or an execution time in units of one minor cycle); there may be additional *operand costs* if some operands cost more to access than others (this is not characteristic of the S/370, but in some architectures the same instruction can operate on either fast registers or memory); and the costs of operand loading (*data movement costs*) are likewise available from the machine description. Whenever an instance of four-byte logical intersection is encountered during code selection, the above three costs are noted or estimated for each of the three possible code sequences, and the one with the least total cost is chosen.

Clearly it is wasteful that this derivation of "best" code sequences is done by the code selector on a case-by-case basis, rather than by the CGG. The code selector should not have to evaluate costs at all; it could work from tables similar in structure to Figure 1-1, in which costs do not appear.

Actually the simple tabular format used in Figure 1-1 is suitable only for choosing among code sequences that are exactly equivalent to each other. Sometimes a choice must be made between code sequences that do not implement exactly the same portions of the source program; an example of this, the choice between using single indexing and double indexing on the DEC VAX-11, is presented in Section 2. To handle such choices, we associated a *benefit* value with each pattern tree, representing an estimate of the cost saved by finding a code sequence to implement that tree rather than, for instance, finding a series of code sequences to implement the smaller subtrees of which it is composed. (A more exact definition of benefit is given in Section 6.) The benefit of a template is in the same units as its base cost (and other costs), but is subtracted from the final total of costs.

For the code selector to be able to get along without costs or benefits, each template must be associated with a specification of the conditions (requirements on the storage allocation of its operands and result) under which it would be used in preference to other templates. If these specifications are to be generated automatically, the CGG must do a comprehensive case analysis examining costs and benefits. For the implementor who takes the path, described in Section 1.1, of doing this and other CGG analysis functions "by hand", it is still useful to start from a machine description augmented with costs and benefits.

From the description of pattern matching given by Cattell,[2, 3] it is easy to spot another source of inefficiency: the pattern matching for each template is separate from the pattern matching for every other template. In our S/370 example, each of the three logical AND instructions is represented by a template with its own tree pattern, but all the tree patterns are identical. When our code selector examines the three templates, it matches each of the tree patterns against the source program tree, getting of course the same

(affirmative or negative) result each time. This is obviously wasteful, but more generally, there is waste any time more than one pattern is matched.

The pattern matcher we used was streamlined, by various *ad hoc* methods, such that little or no effort was wasted in the processing of templates with simple pattern trees in common cases.[15] Rather than describe these methods, however, I draw the reader's attention to a more systematic way of achieving efficiency: the code generator may be organized as a parser (the machine description plays the role of a grammar). The elimination, by the CGG, of the waste described above is similar in spirit to the optimization of a finite-state machine. Ganapathi[5] has demonstrated that it is exactly equivalent to the adaptation of a grammar to the use of "standard context-free parsing techniques (which forbid backup)". He has implemented, in prototype, a code selector approximately equivalent in function to our main code selector, structured as a parser, operating with a machine description structured as an attributed context-free grammar.

2. The PQCC Phase Order

With the PQCC project, we gained additional experience with the three-phase organization of code generation and register allocation that had appeared in the Bliss-11 compiler.[13] A detailed description of this strategy and justification of it are given in RAOC;[10] here, I will only briefly summarize it. A *preliminary code selection* phase (LTN) goes through the motions of code selection, but does not produce a sequence of instructions. Instead it produces a detailed accounting of the temporary storage that would be required for use by those instructions. This accounting takes the form of placeholders called *temporary names* (*TN's*), associated with nodes in the tree representation of the source program. Each TN represents a requirement for one storage location (or contiguous group of locations), possibly but not necessarily restricted to particular types of storage, such as "registers", "accumulators", "odd-numbered registers", or "index registers". Information associated with a TN, accumulated during the code selection pass, includes a *lifetime*, indicating the portions of the program during which the storage location must be reserved for it, and *cost data* from which the importance of finding the "right" location for the TN, relative to the importance of other TN's, can be estimated. Long-lived data items, such as user variables, are also represented by TN's; thus the conventional distinction between "values" and "objects", or between "temporary" and "permanent" storage, which is more or less irrelevant to storage allocation, is camouflaged in a single uniform representation of data items.

The *packing* phase (PACK), which follows LTN, assigns the TN's to particular storage locations. The last phase (CODE) performs actual code selection and generation. It may use its knowledge of what storage location each data item has been assigned to in choosing between equally applicable code sequences. Choices that are made on such a basis clearly cannot be made until after PACK; thus the preliminary code selection

pass of LTN defers those choices, creating TN's on the basis of incomplete knowledge of what storage may be required.

The notion that LTN must defer some code selection choices because it cannot use the results of PACK is relatively novel, and we made some mistakes with it in spite of our previous experience with the three-phase organization in Bliss-11. (The problem seldom arises with the PDP-11 because of the lack of redundancy in its instruction set.) For instance, consider the choice of implementations for integer addition on many architectures, in which there are both general-purpose instructions to add two operands, and special purpose "increment" instructions for use when one of the operands is 1 (or some other known small integer). At first we did not defer the choice between increment and addition instructions; but the example of a code selection problem shown in Figure 2-1, from an example given by Ganapathi,[5] shows that in some cases it must be deferred.

The problem is to implement the statement $a := e1 + 1$, where a is a variable and $e1$ is some expression, on the DEC VAX-11. For each supported width of integers (in the example we use the doubleword or "long" integer instructions), this architecture provides three instructions that perform addition: general-purpose instructions in two-address and three-address format, and an increment instruction in one-address format. In the example the choice between them depends on whether or not the variable a and the result of the expression $e1$ are assigned to the same location.

Figure 2-1 shows the space of possible code sequences, using each of the three applicable addition instructions for both cases of assignment of a and $e1$. The best code sequence for the former case, marked with an asterisk, and the best code sequence for the latter case, marked with a double asterisk, do not use the same instruction. Thus the choice among instructions must be deferred until the assignments of a and $e1$ are known, that is, until after the packing process.

We found cases that pose *phase ordering problems*: the desirability of deferring a choice conflicts with the desirability of providing a complete picture of storage requirements for input to PACK. For instance, one of two competing code sequences may require more or fewer storage locations than the other. An example of this from the IBM S/370 is given in Section 1.2; an even more interesting example, from the DEC VAX-11, is shown in Figure 2-2. (This example is taken from RAOC.[10, ch. 4]) Consider the implementation of the array access $a := b[i + 5]$; assume that the array b is accessed through a pointer to its base, denoted **xb**. The VAX-11 offers both normal indexing and *scaled* indexing, in which the contents of the index register may be multiplied by 2, 4, or 8 before being added to the offset; scaling is useful for access to arrays of 2-byte, 4-byte, or 8-byte elements, since the memory is byte-addressed. In our example, the array b is an array of one-byte elements, so the scaling itself is irrelevant, and if a scaled index register is used in the access to an element of

	a and *e*l assigned to the same location	*a* and *e*l assigned to different locations
Increment instruction	*INCL A	MOVL E1, A INCL A
Two-address instruction	ADDL2 #1, A	MOVL E1, A ADDL2 #1, A
Three-address instruction	ADDL3 A, #1, A	**ADDL3 E1, #1, A

Figure 2-1: Implementations of $a := e1 + 1$ on the DEC VAX-11

b, it is used exactly as an ordinary index (the scale factor is 1). Any operand addressing computation can use both an unscaled and a scaled register; thus, the access to $b[i + 5]$ can use two registers for double indexing, as shown in code sequence a, or can use one register after adding the two indexes, as shown in sequence b. As the two sequences are presented, a is smaller and faster than b, but **xb** and **i** must be in registers for code sequence a, and if they are not assigned to registers by PACK, the cost of loading them for temporary use tips the balance in favor of code sequence b.

a) **xb** and **i** are both assigned to registers;	b) **xb** and **i** are both assigned to memory; **fr** is a free register.
MOVB 5(XB)[I], A	ADDL3 XB, I, FR MOVB 5(FR), A

Figure 2-2: Implementations of $a := b[i + 5]$ on the DEC VAX-11

Note that code sequence b requires a storage location, **fr**, distinct from either **i** or **xb**, while sequence a does not. In deferring the choice between these two sequences, LTN may create a TN to represent FR, or it may refrain from doing so. If it creates the TN, and PACK chooses a register, e.g. R2, for use as **fr**, but then CODE chooses the code sequence that does not use **fr**, then R2 has been wasted, that is, reserved for a purpose for which it is not used. This, in turn, may prevent some other data item from being assigned to R2. The program may use more registers than necessary, or even use more than there are, requiring data to be spilled (unnecessarily) into memory.

On the other hand, if LTN refrains from creating a TN for **fr**, CODE is left to find a free register "on the fly" if it chooses the code sequence that requires one. Both theoretical and practical problems arise: there

may not be any free registers available at that point in the program (a rather unlikely possibility on the VAX-11, but quite likely in analogous situations on some other architectures), and a mechanism must be implemented to allow CODE to carry out on the fly some of the functionality of LTN and PACK.

The above example is rather exotic; many target architectures, after all, offer neither three-address instructions nor double indexing. A related problem is quite commonplace: the problem of deferring the decision of whether or not to create a TN for operand loading. For instance, LTN may recognize that a record access, such as *salary* **of** *employee*, is to be implemented by indexing, but it cannot know whether the index value (the pointer to the record *employee*) will be allocated to an index register, or will have to be loaded into one on the spot. LTN may either create, or refrain from creating, a TN to represent the index register in the loading. A reasonable compromise is to create what is called[10, ch. 6] a *copy TN*, T, with a special relationship with the TN that represents *employee*: PACK recognizes that if *employee* is assigned to a register, T need not be assigned to anything, and if at that point in the packing process T has already been assigned, its assignment may be "undone". In the example of Figure 2-2, there is no such simple relationship between fr and any other TN.

Even when two equivalent code sequences do not have different requirements for TN's, they may lead to different lifetime characteristics for existing TN's. A complete definition of the concept of lifetime is postponed until Section 3; at this point, the nature of the phase ordering problem can be summarized by saying that PACK should have a complete picture of the sequentiality of initializations, updates, and uses of all the data items represented by TN's. For instance, if PACK thinks that the last use of variable *a* occurs (in a straight-line program) before the initialization of variable *b*, it may assign *a* and *b* to the same storage location; but if CODE then chooses a code sequence that reverses the order of these two events, the resulting generated code is almost sure to be incorrect. This disaster can be avoided by using "conservative" lifetime information in PACK, but this can lead to waste of storage.

An example of this problem is shown in Figure 2-3. Both code sequences in the figure compute the difference of *e1* and *e2*, both of which are arbitrary integer expressions. Both sequences leave the result in location t. In code sequence a, the lifetime of t overlaps with that of the value of *e2*, that is, they must not be assigned to the same location. In code sequence b, they don't overlap. The use of conservative lifetime information would prevent *e2* and t from being assigned to the same location; this is necessary for code sequence a, but that sharing of storage might be a useful option when sequence b is to be used.

The above examples show that the three-phase structure for register allocation and code generation is by no means a definitive solution to the fundamental phase ordering problem, which is that each of the processes, register allocation and code selection, gives best results if it runs after the other. But the three-phase structure

Compute *e*1	Compute *e*1
Compute *e*2	Compute *e*2
Load *e*1 into t	SUBL3 E1, E2, T
SUBL2 E2, T	
a) using 2-address subtraction;	b) using 3-address subtraction.

Figure 2-3: Implementations of t := *e*1 − *e*2 on the DEC VAX-11

yields the benefits, in the quality of the code generated, that we have advertised for it,[7, 10] without compromising the efficiency of the compiler (at least by comparison with other compilers that perform global register allocation). Although we learned of problems we hadn't known about before when we began to investigate target machines beyond the DEC PDP-11, serious objections to the overall strategy did not arise.

3. Stages of Operations

If two data items are to share a storage location, they must not both try to make use of it at the same time; updating or creation of one must not destroy the other. To prevent this, we define a *lifetime* for each TN, an enumeration of those portions of the program during which it must have exclusive use of the storage to which it is assigned. In any straight-line segment of the program, the lifetime extends from creations or updates of the data forward to uses of it. The lifetimes of TN's are said to *overlap* if they include in common some point of the program; the TN's are said to *conflict* with each other, and must not be assigned to the same storage locations. The compiler includes a *lifetime analysis* phase, in which the lifetime of each TN is derived from a list of the creations, updates, and uses of the data item and from a graph of the control flow of the program.

The lifetime of a TN is naturally represented as a set of *segments* of the program, each a portion of a *basic block*, that is, a sequence of instructions without branches or jumps. A value *v*, for instance, may be said to be alive "from instruction i_1 to instruction i_2", or even more precisely "from the write-cycle of instruction i_1 to the *n*th read-cycle of instruction i_2". As described in Section 2, lifetime analysis in the PQCC compiler organization is performed before the object instructions are available, that is, before the CODE phase. It is necessary to describe TN lifetimes in terms of the results of the LTN phase; it is desirable to approach closely the precision that would be fully available only after the CODE phase.

For this purpose lifetimes are described in terms of *operations* and *stages* thereof. Each subtree of the program tree that matches an entire code generation template is designated an *operation*, or a single subdivision of "time" for the purpose of lifetime description. These subdivisions are further divided into units by a three-stage scheme, intended to fit any code sequence implementing an expression. If it fits rather

loosely, at least it is to fit in such a way that each instruction of the sequence belongs to exactly one stage and that the data flow implicit in the definition of each stage actually occurs during it. The stages are as follows:

- *operand loading*, including any instructions required to position operands in special-purpose storage, such as accumulators or index registers;
- *result evaluation*, in which the value of the expression is computed;
- *result saving*, including any instructions required to save the value of the expression in order to free special-purpose storage for further use.

Thus the lifetime of value v might be described as, for instance, "from the result saving stage of (the operation represented by) node n_1 to the operand loading stage of node n_2". Knowledge of the simple three-stage model is built into LTN: it follows an equally simple plan for the creation of TN's for each operation, varying only slightly with variations in the structure of the code sequences, and it uses a simple and fixed model of how and when (during which stages) each of the values or objects involved in an operation is accessed. Clearly, this simplicity is conducive to simplicity in the structure of LTN, but it has drawbacks as well. I will explain here some of the shortcomings of the particular three-stage model we used.

One deficiency is the analogue of the deficiency of the simplest model of lifetimes in terms of object instructions. Consider the definition of the result saving stage. In the normal case, for an operation O returning a value, LTN creates a TN to represent the location in which the value is ultimately computed (the *Eval* TN), and a TN to represent the location in which the value is then saved, if saving is necessary (the *Save* TN). LTN records that the *Eval* location is read and the *Save* location is written during the result saving stage of O. Thus, both TN's are alive at that "moment", and they are disallowed from sharing storage. Obviously it would be legal for them to share storage, however; in fact, for many target machines, in which registers for evaluating expressions are plentiful, result saving code is seldom necessary and the two TN's should almost always be assigned to the same storage. Thus it should be recognized that result saving has a "read-cycle" and a "write-cycle", which are two different moments in time, that is, two different stages. More generally, in any simple scheme of stages, there should be enough of them to separate the reading of sources of data movements from the subsequent writing of destinations.

The use of a single "result evaluation" stage is inadequate for different reasons. Early versions of LTN assumed that, except in the case of instructions with target operands (those in which one of the operands is both read and modified), all the operands of an operation are read before the result is written, and hence can share storage with the result. A classic and well-known counterexample is the matrix multiplication operation in many matrix manipulation packages: ordinarily, the reading of the operand matrices is interleaved with the creation of the result matrix in such a way that they must not share storage, and if the package itself does not detect sharing and allocate the necessary temporary intermediate storage, the user must do so.

Matrix multiplication is not typical of the problems faced by optimizing compilers, which historically have been concerned with low-level operations (i.e. operations supported directly by target-machine instructions) and scalar (or very simple vector) objects, but it is a familiar illustration of the principle involved. A humbler example is the computation and storage of a boolean value, such as the result of a comparison, on many target architectures. Here is the skeleton of a code sequence for the statement $b := (e1 \neq e2)$, in which b is a boolean variable and $e1$ and $e2$ are arbitrary arithmetic expressions, for any of several popular architectures:

```
        Compute e1
        Compute e2
        Initialize b to false
        Compare e1 with e2; jump to L if they are equal
        Set b to true
L:
```

In this code sequence b must not share storage with the results of $e1$ or $e2$. This contrasts with the situation for, for instance, three-address arithmetic on the CDC 6600 or the DEC VAX-11, in which the result may share storage with either of the operands. It appears that, for a single model of operations to be general enough to describe both kinds of operators, it would have to be even simpler than the present three-stage model, at a corresponding sacrifice of precision in describing TN lifetimes. Precision could be retained only by the use of different models for different operators, at a sacrifice of compiler simplicity.

4. Data Types

One reason that we chose to implement Ada in later stages of the PQCC project, rather than sticking with simpler languages such as Bliss or C, was that we wanted to explore the effects of common high-level language features on optimization. Particularly interesting language features in this regard are well-supported data types and automatic run-time error checking. As of July 1981, we had identified some of the problems introduced by data types, but most of these had not been solved in the compiler, and the solutions that had been implemented had not been tested to our satisfaction. Here I will discuss some of the problems and proposals for solutions. (I have omitted a discussion of the problems associated with array and record data types, as I was not familiar with that part of the research, nor indeed with the subtleties of the relevant features of the Ada language.)

4.1. Addresses, Integers, and Integer Ranges

Different source-language data types may be represented by the same target-machine data type. A set of optimizations may be applicable if some instructions or instruction sequences that implement operations on one type also implement useful operations on another. This is the case for addresses and integers on some target machines, and likewise for "signed" and "unsigned" integers on some (two's-complement) machines.

Even for languages that do not support explicit manipulation of addresses, we treat addresses as a full-fledged data type, to simplify the processing of array and record accesses. Thus *indexing* is an operation that takes an address and an integer and returns an address, distinguished from but related to *addition*, which takes two integers and returns an integer. On some target machines the same instruction normally used for addition may also be used for indexing. More exotically, on some machines with support in the operand addressing hardware for indexing, the same support may be used for addition. For instance, on the IBM S/370, indexing may be used to add 24-bit integers under some conditions, and on the DEC PDP-10, indexing may be used to add 18-bit integers.

This *simulation* of one operation by another may be impeded or prevented outright by the requirements for, or prohibitions against, run-time error checking. The hardware support for addition may provide automatic trapping for overflow, which must be suppressed if indexing is to be simulated. The hardware support for indexing normally provides no checking whatever for overflow, and hence cannot simulate addition if such checking is needed.

It may happen that two data types are represented differently, but some operations on one type are implementable using coercion of operands to the other type and possibly coercion of the result back to the original type. A set of optimizations is applicable if the coercions are particularly inexpensive, that is, if alternative implementations (with and without coercions) of the same operation may be competitive in cost. This description fits operations such as addition and multiplication on the data types representing integers of different sizes (different ranges) on many target machines.

In general, with addition and multiplication (but not with other operations), an operation on large (wide) operands yielding a result of the same width can be used to simulate an operation on smaller operands. With some other operations, such as division or comparison, such simulation is not so easy but is still possible: after explicit coercion of the operands to the large form (i.e. sign-extension or extension with zeroes), an operation on the large operands will give a result from which the correct small-operand result can be extracted (or, in the case of comparison, will give the same boolean result).

Here is a (probably incomplete) list of what the compiler should know about integers and addresses, and what optimizations it should be capable of discovering:

1. For target machines in which addresses are identical to unsigned integers, the compiler should "know about" (take advantage of) the identity.

2. For machines that support operations on both signed and unsigned integers, the compiler should know about such support.

3. The integer operations of array subscript calculation (multiplication of dimensions and strides, scaling, and addition) should be performed using integers of "appropriate" width. Here are considerations that may influence the choice of width:

    a. Overflow checking of these operations may be unnecessary, especially if there is already checking of subscript bounds (either at compile time or at run time).

    b. The operands and results can frequently be guaranteed to be all non-negative.

    c. Operations on short integers are usually less costly than operations on wide integers.

4. The compiler should know if indexing can be simulated by addition.

5. The compiler should know if indexing can simulate addition. Also,

    a. For machines in which this simulation is available only for certain special cases of addition (e.g. the DEC PDP-10 and the IBM S/370), the compiler should recognize those special cases.

    b. If there are language or implementation requirements for overflow checking, and if there is hardware support for such checking, the compiler should take into account any effect this has on the simulation (e.g. addition is checked but indexing is not).

6. The simulation of short-integer operations by wide-integer operations should be routinely used whenever it is the best (or only) implementation of the short-integer operation. This includes:

    a. direct simulation, as with addition and multiplication;

    b. simulation after explicit coercion of operands, as with division and comparison;

    c. recognition that certain instructions, such as ADDI or MULI on the DEC PDP-10, can always be used for short-integer operations, though they can be used for wide-integer operations only for restricted classes of operands.

Items 1 and 2 on this "wish list" may seem trivial, but they impose significant requirements on the machine description language. Signed and unsigned operations must be distinct; thus, for instance, the version of ISP exemplified by the description of the CDC 6600 given by Bell and Newell[1] is too vague, for it does not reveal whether the comparisons available between 18-bit integers in that machine are signed or unsigned (an important question since addresses are unsigned 18-bit integers). Short and wide versions of an operation must likewise be represented by different names, as context is not always sufficient to identify them. Even knowledge of operand types is not always sufficient. Some architectures offer two different versions of integer multiplication, which do not differ in the width of the operands required, but differ in that one preserves the whole double-width result while the other throws away the top half; these two operators must have different names in the machine description.

Item 3 of the wish list deserves to be clarified by an example. Consider an access to a 10x10 array of words on a DEC PDP-10: $a[b, c]$. The address arithmetic to be performed for this access, in one implementation of arrays, appears to the compiler as (using BLISS dot notation*):

$$(a - 1) + (.b * 10) + .c$$

The indexing, scaling, and addition in this expression could be performed by the 36-bit signed integer arithmetic instructions that would normally be used, on the PDP-10, to evaluate source-language integer expressions. But since the end result will be used as an 18-bit PDP-10 address, all the operations should be performed using the hardware support for indexing and for cheap 18-bit unsigned integer arithmetic. In the similar array access $a[b, c + d]$, even the source-language addition could be evaluated using the special hardware, if the need to check it for overflow could be obviated by compile-time analysis.

We found it easy to suggest extensions to the machine description language to describe the various relationships between operators, and to specify corresponding procedures in the compiler to use the extensions, thereby satisfying in a piecemeal fashion each of the items on the wish list. It would be desirable to have a single notation throughout the machine description language for describing overflow and overflow-checking features. Historically, the identification of the problems on the wish list occurred too late for us to implement most such suggestions, although the total task does not appear to be unreasonably large.

5. Preference and Weak Preference

The sharing of storage by two data items may allow the code for initializing one of them to be omitted. Frequently, for instance, the value of an expression is initialized from one of its operands, and code can be saved if the operand and the value both use the same storage. So that the packing process can take such possibilities into account, an earlier phase (LTN) accumulates a record of all the data movements anticipated in the object program, called the *preference function*:

> [*Preference costs* are] the costs of moving data around in storage, either because of assignments in the user program, or because of requirements for special-purpose locations.[10, p. 5]

> Let T be the set of all TN's; the *preference relation* is a relation on T x T. One TN is related to another under preference if a data movement between the locations they represent is anticipated.... Actually, it is more accurate to speak of a preference *function*, a mapping from T x T to the set of possible costs.[10, p. 59] The value of the preference function for ... two TN's [is] the total cost of the anticipated data movements.[10, p. 122]

In practice the preference function incorporated costs that might not be implied by a strict interpretation of the above definition. The value of the preference function for a pair of TN's included, not just the costs of

---

* In the BLISS language,[12] the dot (period) denotes the "dereferencing" operator; thus for variables v and w, $v + .w$ denotes the sum of the *address* of v with the *contents* (*value*) of w.

direct data movements, but any costs that could be eliminated by assigning the TN's to the same storage location. To clarify this distinction I will discuss the preference costs associated with the evaluation of an integer addition, *e*1 + *e*2, on three different target machines. The names *e*1 and *e*2 denote arbitrary expressions.

Figure 5-1 shows two possible code sequences for evaluating *e*1 + *e*2 on the DEC PDP-11. L1 and L2 denote the locations in which the values of *e*1 and *e*2 were previously computed.

```
ADD     L2, L1                  MOV     L1, L0
                                ADD     L2, L0
```

a) L1 is re-used b) L1 is not re-used

Figure 5-1: Implementations of *e*1 + *e*2 on the DEC PDP-11

In code sequence a, location L1 is re-used to hold the sum; in sequence b, L1 is not re-used, and a third location, L0, is used instead. Code sequence b is the more expensive of the two locally; the difference in cost is the cost of the data movement, the instruction to load L0. Thus T1, the TN representing *e*1, and T0, the TN representing the sum, are related by preference; this is the "classic" preference relation defined in the above-cited passages. The value of the preference function is the cost of that instruction: the two bytes of instruction space it takes up (assuming L0 is a register), or the processor and memory cycles it requires at execution time (possibly weighted by its expected frequency of execution).

Figure 5-2 shows two exactly analogous code sequences for the DEC VAX-11. As in the previous example, L1 is re-used in code sequence a but not re-used in sequence b.

```
ADDL2   L2, L1                  ADDL3   L1, L2, L0
```

a) Two-address instruction b) Three-address instruction

Figure 5-2: Implementations of *e*1 + *e*2 on the DEC VAX-11

Again, sequence b is more expensive, but here there is no explicit data movement code on which to "blame" the added cost. The TN's, T1 and T0, are related under *weak preference*. We treated the situation as if there were a preference relation: the value of the preference function for the two TN's is the difference in cost between the two code sequences. (The cost of the extra operand access is one byte of instruction space, or one memory access at execution time.)

Figure 5-3 shows analogous code sequences for the Livermore Labs S-1.[6] Here there is not even a clear cost difference between the two code sequences. In sequence b the sum must be produced in one of two particular registers, denoted RTA and RTB, and this restriction could have some influence on global allocation and costs. Locally, however, sequence b is not more expensive than sequence a.

ADD.S L1, L2	ADD.S RTA, L1, L2
a) Two-address instruction	b) Three-address instruction

Figure 5-3: Implementations of *e*1 + *e*2 on the LLL S-1

It is difficult to suggest a straightforward way of describing this situation in terms of preference costs or a preference function. The practical reflection of this theoretical problem is that it is not clear just how the packing algorithm described in RAOC[10, ch. 10] should take into account the very weak preference relation between T1 and T0 in cases like this.

6. Access Modes

Many target architectures include non-trivial operand addressing mechanisms, such as indexing, scaling, and indirection. Conventionally, these features are meant to be used to evaluate "addressing expressions", as the instructions themselves evaluate arithmetic or logical expressions, but the boundary between the two is sometimes blurred, as observed in Section 4.1. In our machine descriptions, each legal combination of addressing mechanisms, including relatively simple ones such as direct access to registers or immediate access to constants, is represented by an *access mode* (*AM*), a pattern tree similar to the pattern trees that describe the effects of instructions. In principle, opportunities to use addressing hardware can be discovered and evaluated by the same pattern-matching mechanism that we use to find opportunities to use instructions. In the PQCC system, however, we extracted AM processing from both LTN and CODE and put it in a separate (preprocessing) phase, AMD.

This was intended to simplify LTN and CODE. Because the complete results of the preprocessing must be recorded from one phase to another, however, different algorithms were introduced for the preprocessing phase, possibly increasing, not decreasing, the complexity of the system as a whole. Certain optimization strategies were also to have been promoted, as claimed in RAOC,[10, ch. 4] by the separation of AMD as a phase by itself, but we do not have any data on the practical value of these strategies.

AMD records the results of the analysis that it performs by associating with each program tree node an undifferentiated set (represented as a bit mask) of the AM's that could be used when the value represented by

the node is used as an instruction operand. It was intended that CODE should select from this set on the basis of the costs and benefits associated with the AM's, plus loading costs and other relevant data in the machine description. We found, however, that cost is not the only criterion for selecting AM's from the set, and the "undifferentiated set" does not provide enough information to make the right selection in all cases. Consider the following rather unusual implementation of the assignment $a := b$, in which a is a statically allocated or "own" variable, on the DEC PDP-11:

```
MOV   #A, R0  ; load the address of a into a register
MOV   B, @R0  ; modify a by indirection through the register
```

Of course, the normal implementation of the assignment would not use an intermediate register, but the implementation given here would be useful if it were desirable to have the address of a in a register at some later point in the program. Note that in this code sequence, both operands of the first instruction, and one operand (the second) of the second instruction, all refer to the same node in the program tree, that is, to the reference to a. Yet the three operands use different AM's, and CODE must be careful to use the three AM's exactly as given here and in the correct order. Thus, the cooperation between AMD and both LTN and CODE must be somewhat closer than what we had envisioned.[10, ch. 4]

It is appropriate to discuss the estimation of "benefits" of tree patterns in this section, since we designed and implemented a realistic model of benefits for AM's, but never had such a model for instruction pattern trees. As mentioned in Section 1.2, the benefit of an AM, such as indexing, is the cost of the code saved by using it instead of using, for instance, less powerful AM's, such as indirection, or by foregoing the use of any but the very simplest (direct-access and immediate-access) AM's, as when explicit shifting or multiplication is used instead of automatic scaling. It is hard to apply this rather vague definition to define an absolute benefit for a particular AM, but we can build a table of benefits from consideration of the *relative* benefits of competing AM's. For instance, the difference between the benefits of the single and double indexing AM's used in the example in Figure 2-1 should be the cost difference between the code sequence using one AM and the code sequence using the other, *under identical (and ideal) conditions of register allocation*, i.e. when **xb** and **i** are both assigned to registers. The base on which the tower of relative benefits rests must be a foundation of absolute benefits, and these are defined in an obvious and intuitive way be assigning benefit 0 to the AM's that represent direct access to the various different classes of registers and memory.

Since operand loading costs cannot normally be known before the packing process has completed, the selection of AM's in early versions of LTN was strictly benefit-driven. The algorithm presented by Cattell[2, p. 37] as the "Maximal Munching Method" also has this character. The application of purely benefit-driven strategy to AM selection is described in detail in RAOC.[10, p. 82] The algorithm given there never assumes that an operand will use a simple direct-access (called "molecular") AM if a more exotic ("compound") AM is available instead. This strategy is unsatisfactorily simple-minded; even as early as the

LTN phase some estimate of operand loading costs must be made if a realistic description of storage requirements is to be passed to PACK.

An example of the problems that arise from a too straightforward application of the maximal munch rule is the handling of a typical access to a variable, *v*. On many or most architectures this can be done either by a direct access, or by loading the address of *v* into a register followed by an indirect access (through the register). The indirect-access AM has a higher benefit than the direct-access AM; to use Cattell's terminology, it munches more of the program tree. Unless the address of *v* is already available, however, the cost of loading it outweighs the benefit difference, and direct access is the "normal" way to access *v*. If LTN fails to take this cost consideration into account, it invariably assumes that indirect access will be used, and creates a TN to represent the location into which *v* will be loaded. As discussed in connection with Figure 2-2, the allocation of these extra TN's causes storage to be wasted and misused.

The July 1981 version of LTN, and indeed Cattell's earlier code generator,[4] were patched so that at least simple variable accesses were treated correctly. More generally, LTN must be prepared to assume the necessity for operand loading in some situations, and to take its cost into account in the selection of AM's.

Summary

In view of the motivation and even the name of the research project, perhaps the most important issue discussed in this paper is the use of compiler generation tools. It is disappointing that we did not go farther with the techniques of automated search and analysis than Cattell had gone. Even the primitive CGG that we used, however, enabled us to keep target-architecture knowledge in a single machine description, while using it in more than one phase. The significance of this can best be appreciated by the compiler writer who has attempted to keep up parallel maintenance of separate target-architecture knowledge sources (code or tables) for separate phases.

Next in importance after the questions of compiler-compiler structure must come those of the compiler phase structure, discussed in Section 2. There is a fundamental phase ordering problem involving register allocation and code selection, in that either can produce more nearly optimal results if the other has already run. Our three-phase approach to this problem should not be regarded as a "solution" in the sense of definitively making the problem go away. It is simply a practical structure for addressing the problem, which can be adapted to small or large compilers (by adjustment of the size and ambition of the packing phase), and which leaves room for further experimentation in code generation and register allocation techniques.

The problems discussed in the later sections of this paper are not quite so fundamental. It should be clear from the discussion of data types and access modes that we have only scratched the surface of a systematic

treatment of these areas. It is to be hoped that their importance will be more widely recognized in the current resurgence of interest in retargetable optimizing compilers.

References

[1] C.G. Bell and A. Newell.
Computer Structures: Readings and Examples.
McGraw-Hill, 1971.

[2] R.G.G. Cattell.
Formalization and Automatic Derivation of Code Generators.
PhD thesis, Carnegie-Mellon University, April, 1978.
Available as Technical Report CMU-CS-78-115.

[3] R.G.G. Cattell, J.M. Newcomer, and B.W. Leverett.
Code Generation in a Machine-Independent Compiler.
In *SIGPLAN Conference on Compiler Construction*, pages 65-75. ACM, Denver, 1979.
Published in SIGPLAN Notices, August 1979.

[4] R.G.G. Cattell.
Personal communication.

[5] M. Ganapathi.
Retargetable Code Generation and Optimization Using Attribute Grammars.
PhD thesis, University of Wisconsin, December, 1980.

[6] B.T. Hailpern and B.L. Hitson.
S-1 Architecture Manual.
Report No. 161, STAN-CS-79-715, Stanford University, Department of Electrical Engineering, January, 1979.

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

[8] 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.
Technical Report CMU-CS-79-105, Carnegie-Mellon University, Computer Science Department, February, 1979.

[9] 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.

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

[11] J.D. Oakley.
Symbolic Execution of Formal Machine Descriptions.
PhD thesis, Carnegie-Mellon University, April, 1979.

[12] W.A. Wulf, D.B. Russell, and A.N. Habermann.
BLISS: a Language for Systems Programming.
Communications of the ACM 14(12):780-790, December, 1971.

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

[14] 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.

[15] W.A. Wulf.
Personal communication.

Acknowledgments

I am grateful for valuable advice and corrections from Bill Wulf and Ric Cattell.

I have also benefited from conversations with various persons at Digital Equipment Corporation's Software Engineering Facility, including Steve Hobbs, and persons (from Siemens, CII-Honeywell Bull, and Alsys) engaged in the European Ada Compiler Project, including Olivier Roubine.

It is intended that this paper should be only one of a series of papers summarizing the results of the PQCC project. However, this is as good a place as any to give a complete acknowledgment of project participants, not only those who worked in the areas of code generation and register allocation. Persons on the following list either (1) have made substantial contributions to the project but have not been acknowledged in previous papers, or (2) were participating in the project at the time of the July 1981 demonstration: Margaret Beard, Ellen Borison, Ben Brosgol, Reidar Conradi, David Dill, Regis Hoffmann, Dan Johnston, Paul Knueven, David Lamb, John Nestor, Joe Newcomer, Andy Reiner, David Stryker, and Bill Wulf.

I. Summary of Phase Performance

The 30+ implemented phases of the July 1981 compiler were combined, as it was linked/loaded, into eight sequential phases, of which all but the output lister are summarized here. All measurements were done with full support debugging turned on. This has been measured as increasing the size by a factor of 2 and the time taken by a factor of 4.

Phase	*Code size*	*Data size*	*Speed^{-1} (μs/node)*	Description
Front end	14999	7220	9400	Parses; constructs tree
CWVM	27722	24521	3259	Converts Ada constructs to TCOL
Flow	17410	2007	7200	Does global data flow analysis
Delay	11316	7813	4190	Does pre-CG optimizations and AMD
MDX	1442	730	581	Converts Ada operators to those in MD
TNBind	21609	2355	~	LTN plus PACK
Code	9600	8524	4913	Code selection and peephole optimization

II. Selected Extracts from a Machine Description

Here I will show some selections from the description of the DEC VAX-11, the one intermediate-format machine description that had been used and debugged as of July 1981. I will only give selections from sections of the description that are relevant to this paper, that is, the code selection templates ("productions") and the access modes. The machine description also included descriptions of the storage structure (*storage bases* and *storage classes*, as described elsewhere[2, 10]), and a means of describing which access modes can be used for each instruction operand (*operand classes*), but a presentation of these would be outside the scope of this paper.

Figure 6-1 shows three code generation templates. Each consists syntactically of a name (optional) followed by a sequence of attributes and their values, delimited by colons and semicolons in an obvious way. Using the terminology of Section 1, the *pattern* in each template is the value of the **pattern** attribute, and the *code sequence* is the value of the **action** attribute. The following commentaries may facilitate the understanding of Figure 6-1:

- All three templates describe operations on 32-bit integers. The customary operator names ("+" and "NEQ") in the patterns are given a suffix ("i") by convention to distinguish them from similar operators on 8-bit, 16-bit, and 64-bit operands.

- The first template named ADD describes two-address addition; the second describes three-address addition. Their pattern trees are identical, the only difference between their **pattern** attribute values being the presence in the former template of a predicate, **same01**, evaluated separately from the pattern-matching process.

- The **type** and the boolean **Notarget** attributes are among the "cryptic notations" mentioned in Section 1.1, describing features of the pattern and/or code sequence that could in principle be derived from inspection of them by an analysis program. The **type** attribute gives the *schema* in which the template is to be classified. Schemas, described by Cattell[2] and in RAOC[10], are beyond the scope of this paper. The **Notarget** attribute is relevant only for the last two templates; its presence signifies that none of the operands of the pattern tree is a *target operand*, as defined in

```
production NEQi
        type:           flow;
        pattern:        $1:RL NEQi $2:RL;
        action:         emit cmpL, $1, $2/
                        emit bneq, $T/
                        emit br, $F;
        space:          7;
        time:           49;
        benefit:        3;
end production

production ADD
        type:           value;
        pattern:        $0:WL := $1:RL '+i' $2:RL | same01;
        action:         emit addL2, $2, $0;
        space:          3;
        time:           49;
        benefit:        5;
end production

production ADD
        type:           value;
        pattern:        $0:WL := $1:RL '+i' $2:RL;
        action:         emit addL3, $1, $2, $0;
        NoTarget;
        space:          4;
        time:           49;
        benefit:        5;
end production
```

Figure 6-1: Code Generation Templates

Section 1.2.

- The **space** and **time** attributes were intended to describe costs. The former was actually used, and gives the size of the code sequence in bytes; the latter was not even filled in correctly, and the value "49" was used uniformly throughout to make this obvious. If both attributes had been implemented (i.e. if the **time** attribute were an execution time in units of, for instance, one processor cycle), the compiler writer could specify a function to be used to compute the overall cost of the code sequence from these two attribute values. In keeping with this philosophy, the *benefit* of the template should have been described by two attributes, but only one is present, and the value with which it was filled in, which happens to be the number of nodes in the pattern tree, is wildly inappropriate as a starting-point for computation of a genuine benefit figure. As mentioned in Section 6, we never did design a realistic model of benefits for instruction pattern trees.

Figure 6-2 shows two access modes. They use an attribute-value syntax similar to that of the code generation templates. It is striking that there are no pattern trees: AM processing, having been extracted from LTN and CODE into a separate phase, uses algorithms (sketched in RAOC[10, ch. 4]) that are quite

```
am B<R>[R]_a_b
        size: 8;
        benefit: 9;      ! {Add13 B<R1>, [R2], R3}
        sc: M_b_SC;
        format: B<r>[R]_A1;
        space: 2;
        time: 0;
        type: interesting;
        parms: $0: R_v_1 L, $1: R_v_1 R L, $2: I_b R R;
        transforms
                {R_v_1} add_op { B<R>_a_b } ;    !waw mod
        end transforms;
end B<R>[R]_a_b

am B<R>[R]_v_b
        size: 8;
        benefit: 12;     ! {Add13 B<R1>, [R2], R3; Mov1 @R3, R4}
        sc: M_b_SC;
        format: B<R>[R]_A1;
        space: 2;
        time: 0;
        type: interesting;
        parms: $0: R_v_1 D L, $1: R_v_1 D R L, $2: I_b D R R;
        transforms { B<R>[R]_a_b } fetch_op;
        end transforms;
end B<R>[R]_v_b
```

Figure 6-2: Access Modes

different from the usual pattern-matching and that do not use the pattern tree directly, but use information abstracted from it. This information appears as the **parms** and **transforms** attributes; a full explanation of these is outside the scope of this paper. The abstraction of this information would ordinarily be the task of the analysis program.

Both AM's describe double indexing with scaling, using an offset that is one byte wide, to access a quantity that is one byte wide. (By convention, the capital "B" in the name of the access mode denotes the width of the offset, and the lower-case "b" denotes the width of the data to be addressed.) The first AM is for use by instructions that use only the address computed, while the second is for accessing the memory location at that address. The attributes are explained as follows:

- The **size** is the width (in bits) of the data addressed.
- The **space** and **time** are costs (**time** is not implemented, just as in the code generation templates), and the **benefit** is a benefit, computed by a method related to (i.e. an earlier version of) that described in Section 6. The comment after each benefit value is the code sequence, or substitute method of "simulating" the effect of using the access mode, from which the benefit of the access mode was derived.

- The **format** is a template for output. Angle brackets are used in place of parentheses for obscure reasons. The **sc** attribute specifies the *storage class*, in this case memory (byte-addressed), of the data being addressed. The **type** attribute, like the attribute of the same name in code generation templates, is a classification of the access mode by which the set of access modes is sorted and searched. Discussion of these attributes is beyond the scope of this paper.