A retargetable vector code generator

by TOM C. REYES*

Oklahoma State University
Stillwater, Oklahoma

ABSTRACT

This paper describes the prototype implementation of a retargetable vector code generator which applies recent advances in automatic (scalar) code generation techniques to the task of generating code for a vector source language. The source language is a subset of Fortran 8X, the proposed successor to the Fortran 77 standard. The target machine is an attached vector processor. This work extends Cattell's maximal munch method to vector code generation. Variants of TCOL are used for the intermediate representation of both the source program and the target machine description table. The prototype implementation demonstrates the feasibility of template-driven vector code generation and emphasizes the importance of code optimization in a vector compiler.

* The author's present address is: Amoco Production Company Research Center, P.O. Box 3385, Tulsa, OK 74102
INTRODUCTION

Most vector processors are programmed using either a low-level machine-specific language or a scalar high-level language (usually Fortran 77) with a vectorizing compiler. Code written in a low-level language is inherently nonportable. Writing good efficient code in a low-level language is also a difficult and often expensive task. Pure Fortran 77 code is portable but inherently sequential, obscuring the parallelism present in the application program and requiring a vectorizing compiler to rediscover vector operations in sequential loops. In practice, most Fortran programs which are run on vector machines are not completely portable either. It is not uncommon to maintain multiple versions of Fortran programs which are regularly run on different machine and compiler combinations.

The recent development of compilable vector languages such as Fortran-8X,2 Vector-C,3 Actus,4 and modified APL5 has made it possible to write scientific code using a more concise and higher level notation. The parallelism inherent in array operations can now be directly expressed in these languages obviating the need for vectorization of scalar code. The implementation of these languages on vector machines presents some new, interesting problems. Besides generating good vector (and scalar) code, it is also important for a vector language compiler to be retargetable to other new and existing vector processors. A family of retargetable vector compilers which implements the same language on different machines and shares a common machine-independent core is more likely to enhance program portability than a collection of vectorizing compilers. Recent advances in compiler technology, particularly in the automatic code generation area, have largely focused on scalar machines and scalar languages.6,7,8,9,10 Previous vector language implementers3,4,5 have taken an approach to retargetable code generation similar to that used in the Pascal P-compiler, producing code for an abstract machine and then expanding this virtual code into the real code appropriate for the target machine. A major problem with this approach is the fundamental conflict between the need for a truly machine-independent abstract machine model and the need to fully exploit the power of a particular target machine.

A more promising approach, which allows the realization of the optimization goals of a vector compiler without sacrificing compiler retargetability, is pattern-matched vector code generation. This paper describes the application of a template-driven code generation technique to the task of implementing a vector language on an attached vector processor. This work extends Cattell's tree-matching algorithm11 to vector code generation.

SOURCE LANGUAGE

Our source language is a subset of Fortran-8X, the proposed successor to the Fortran-77 standard. Fortran-8X is a superset of Fortran-77 and has a rich array processing facility among its set of new features.

Fortran-8X's Array Processing Facility

Fortran-8X's array processing features are designed to allow the programmer to implement numerical algorithms in a more concise and natural notation. For example, the following Fortran-8X code may be used to compute the variance of n sample points \( n > 1 \) contained in a vector \( x \) (of size \( n \)):

\[
\text{variance} = \frac{\text{SUM}((x - \text{SUM}(x)/n)\times2))/(n - 1)}
\]

Note how clearly this single statement indicates what is being computed without the distraction of an extraneous loop and loop index variable which in Fortran-77, and in other scalar languages, would have been required to index thru the vector \( x \).

Fortran-8X carries even further the idea of allowing whole array operations by extending all scalar operations to apply element-wise to any conformable operands. An element-wise operation applies a scalar operation on the corresponding elements of its operands to produce the corresponding element of the result array. These element operations can be performed in any order simultaneously.

Fortran-8X allows subsets of an array, called array sections, to be used as operands. An array section is an array itself and is the result of subscripting a parent array with one or more non-scalar subscripts and zero or more scalar subscripts. An array section is the set of elements from the parent array selected by the non-scalar and scalar subscripts. One way of specifying non-scalar subscripts is by triplet notation which consists of three scalar integer expressions separated by colons, namely:

\[
\langle \text{array} \rangle \langle \text{begin} \rangle : \langle \text{end} \rangle : \langle \text{stride} \rangle
\]

This triplet notation represents a sequence of integer subscript values starting from \( \text{begin} \) to \( \text{end} \) in increments of \( \text{stride} \).

Fortran-8X provides many other array processing features including array constructors, dynamic arrays, and masked array assignments. However, we chose not to complicate our prototyping efforts with these features.
TARGET MACHINE

Our target machine is the IBM 3838 attached vector processor. Although the 3838 has been rendered largely obsolete by more recent vector processors, it provided us almost unlimited and very inexpensive access to a real target machine with vector instructions. A vendor-supplied interface called the Vector Processing SubSystem (VPSS) provided 40 memory-to-memory vector instructions, 21 scalar arithmetic and branching instructions, and 13 housekeeping operations for programming the 3838.

COMPILER STRUCTURE

Our compiler design and choice of intermediate representation were heavily influenced by Intermetrics’ compiler projects. The Intermetrics projects were, in turn, inspired by W. Wulf’s Production Quality Compiler-Compiler project at Carnegie-Mellon University.

We greatly simplified our compiler design by deliberately deferring inclusion of any optimization phases until the feasibility of a retargetable vector code generator was established. Our compiler design is diagrammed in Figure 1.

No attempt was made at implementing a compiler front-end to process Fortran-8X statements. We simply hypothesize the existence of a Fortran-8X scanner, parser, and semantic analyzer. We also assume the existence of an array-to-vector transformation phase (labeled Vectorizer in the diagram) which unravels multi-dimensional array expressions and assignments into one or more nested loops, bracketing the same expressions and assignments but operating only on vector and scalar operands.

INTERMEDIATE REPRESENTATION

Two intermediate languages, both variants of TCOL, are used in our compiler design:

1. TCOL-8X is the output of a Fortran-8X compiler front-end and the input to the back-end.
2. TCOL-3838 is the intermediate representation used in the back-end.

TCOL-8X

TCOL-8X is the high-level Fortran-8X-specific and machine-independent notation used in the output of the compiler front-end. TCOL-8X closely reflects the structure and the semantics of Fortran-8X constructs. A TCOL-8X VECTOR node reflects the structure and the attributes of a Fortran-8X array reference which it represents. It references
a parent ARRAY node and a sequence of SCALAR, TREE, or SECTION nodes representing the subscripts. It does not expose implicit machine-dependent address computations. Similarly, a TCOL-8X SCALAR node models the attributes of the Fortran-8X scalar variable it represents. It has no storage location attribute because storage binding is a machine-dependent operation which is deferred to the Storage-Binding/Expansion phase. Fortran-8X control constructs retain their basic structure in TCOL-8X. A Fortran DO-loop is represented in TCOL-8X as a loop-control part and a loop-body. It is not decomposed into conditional and unconditional branches which are often machine-dependent (e.g., some machines have a “do-loop” instruction).

TCOL-3838

In contrast to the machine-independence of TCOL-8X, TCOL-3838 closely reflects the data types, operations, addressing modes, and instruction formats available on our target machine.

The main motivation for the low-level nature of TCOL-3838 notation is the need to represent the low-level semantics of both program constructs and machine instructions in the same notation. The low-level semantics of program constructs are represented as TCOL-3838 program trees. The effects of executing a target machine instruction are represented as a TCOL-3838 pattern tree. This uniform representation of program trees and pattern trees allows the instruction selection problem of code generation to be recast into a pattern matching problem. If the TCOL-3838 program tree representing the low-level semantics of a particular program construct matches the TCOL-3838 pattern tree representing the effects of executing a particular 3838 machine instruction, then the corresponding target machine instruction can be emitted to implement the program construct.

The TCOL-3838 VECTOR node type closely models the target machine representation of vector operands. A TCOL-3838 VECTOR node has four attributes: a base, a count, a stride, and an operand class. The base attribute references another TCOL-3838 structure which represents the computation of the address of the first vector element. The count attribute references another TCOL-3838 structure which represents the computation of the number of elements in the vector operand. The stride attribute references another TCOL-3838 structure which represents the distance between consecutive elements of the vector operand. The operand class attribute represents the vector operand access modes of the target machine. TCOL-3838 completely exposes all operations that are implicit in the TCOL-8X notation including the arithmetic operations involved in determining the number of iterations in a general Fortran DO-loop.

To completely specify a target machine instruction for code generation purposes, the description of the effects of the instruction must be associated with a code generator action to emit the properly instantiated instruction according to a specified instruction format. TCOL-3838 is also used for describing these code generation actions.

VECTOR CODE GENERATION

The code generation phase is essentially a three-step operation:

1. A pattern matching step searches a target machine description table for instruction templates to match against the program tree. The goal of this step is to find an instruction template which matches the program tree structurally and semantically.
2. A pattern instantiation step fills in the “holes” of the instruction template with values obtained from the program tree. These “holes” in the instruction template correspond to the memory addresses, the register numbers, and the literal values of the instruction’s operands.
3. An action sequence executes the code generation directives specified by the instruction template. These directives may call for:
   a. The recursive invocation of the code generator to generate code for a program subtree or to instantiate an operand of a vector instruction
   b. The writing out of a fully instantiated instruction into the output object file
   c. The creation of a statement label
   d. The return of the instantiated values of a vector operand’s base, count, and stride

The code generation algorithm, as implemented in our compiler back-end, is essentially Cattell’s maximal munch method. Published applications of maximal munch seem to have been directed exclusively to scalar code generation. We did not have to extend Cattell’s maximal munch method in order to use it for vector code generation. The three-step pattern-matching/instantiation/action-sequence model is equally applicable to both scalar and vector code generation.

We did extend the TCOL intermediate representation (TCOL-3838) to reflect the vector operand addressing modes and instructions available on our target machine and to represent the four kinds of code generation directives in the action-sequence step. Correspondingly, our target machine description table has two new sections devoted to vector instruction templates and to vector operand access mode templates respectively. The code generator was also extended to interpret the new code generation directives specified by these templates.

Just like other maximal munchers, our code generator knows about partial matches which can be transformed into complete matches by fetch or store decomposition. When the offending operand is a vector source operand, vector fetch decomposition obtains a vector temporary and conceptually creates a new program subtree to move the offending source operand into the required vector operand storage and recursively invokes the code generator on this new subtree. Having fixed the offending source operand(s) in a partially matching template, the code generator can then execute the template’s code generation directives and fix any offending destination operand(s) by store decomposition. In a simple one-pass code generation scheme such as ours, the allocation and deallocation of vector and scalar temporaries associated with vector
and scalar fetch, and store decomposition, is performed during the single code generation pass. In a multi-pass optimizing compiler, the binding of temporaries to program tree nodes is done in a separate pass.

TARGET MACHINE DESCRIPTION TABLE

We have not attempted to automate the creation of the target machine description table. Instead, we simply wrote the instruction templates by hand.

Organization of the Machine Description Table

The target machine description table is organized for efficient access by the code generator. It consists of five major sections:

1. Scalar arithmetic instruction templates
2. Vector arithmetic instruction templates
3. Vector operand class and access mode templates
4. Control construct templates
5. Conditional branching templates

Within each section, templates whose pattern trees represent similar instructions (e.g., "add" and "add-immediate") are sorted by increasing cost per number of node.

Completeness

The completeness of the machine description table is an important issue in retargetable code generators. Code generator blockage is usually a symptom of an incomplete or incorrect machine description table. Tools for writing code generators find their most important use in this area of assuring completeness in the machine description table.

Due to our lack of such a tool and our rather specialized target machine, we could not completely cover all legal program constructs that can be generated by a Fortran-8X compiler front-end. We had to restrict our source programs precisely to those programs whose computations can profitably be off-loaded to our target machine.

Exotic Instructions

On our target machine, about half of the 40 algorithmic vector operations can be considered exotic instructions. Examples include fast Fourier transforms, vector square root, and vector tangent. We did not write any templates for these exotic instructions because of their limited applicability to the translation of general-purpose Fortran-8X constructs.

Despite their specialized nature, these exotic instructions can be used to implement the "high-level" operations they were designed for. Equally exotic TCOL-3838, and TCOL-8X, operators and structures can be defined to model the semantics of these fancy instructions. Adding templates using these exotic operators to our machine description table and extending our source language by predefining these fancy operations as intrinsic operations, would allow our compiler to provide the user all the native capabilities of our target machine.

AN EVALUATION OF THE COMPILER BACK-END

To establish a reference point for comparing the quality of the output code of our compiler back-end, we hand assembled and optimized some of the Livermore kernels into Vector Processing SubSystem CALLS. The output from the execution of these two versions of the test routines were compared and found to be identical, giving us some confidence that our compiler back-end is generating correct code.

A static analysis of the compiled code against the hand-crafted code yielded the results shown in Table 1. Our compiler back-end generates only vector arithmetic instructions which are absolutely necessary. However, it also generates too many unnecessary scalar instructions. A large proportion of these redundant scalar instructions are loads and stores.

A closer examination of the compiled code indicates that most of the redundant loads and stores arise from the allocation and deallocation of vector, and scalar, temporaries during vector, and scalar, fetch decomposition. On-the-fly vector, and scalar, temporary management is the main culprit. Some scalar instructions are redundant because they constitute common subexpressions or invariant computations which can be eliminated or moved out of loops. Our design decision to exclude any optimization phases in our compiler back-end has come back to haunt us.

To measure the execution efficiency of the compiled code against the hand-crafted code, we timed the execution of the compiled code and the hand-crafted code on an essentially unloaded IBM 3090 with a similarly unloaded 3838 attached processor. The benchmark code is Livermore kernel 8 which is a fragment of a P.D.E. integration routine. We performed two sets of timing runs for two different size settings of the table.

<table>
<thead>
<tr>
<th>Table I—Static analysis of compiled vs. hand-crafted VPSS code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of VPSS Calls</td>
</tr>
<tr>
<td>C Compiled</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>Setup</td>
</tr>
<tr>
<td>Transfers</td>
</tr>
<tr>
<td>Housekeeping</td>
</tr>
<tr>
<td>Scalar Arithmetic</td>
</tr>
<tr>
<td>Scalar Moves</td>
</tr>
<tr>
<td>Branches</td>
</tr>
<tr>
<td>Scalar Operations</td>
</tr>
<tr>
<td>Real Arithmetic</td>
</tr>
<tr>
<td>Vector Operations</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>
array operands. For each set, we isolated data transfer times from actual target machine computation times. The results are shown in Table II.

The run-time numbers are worse than those for code size because most of the redundant scalar instructions happen to be inside loops. The megaflop rates indicate that our target machine executes scalar and simple vector instructions with short operands very inefficiently compared to the exotic vector instructions (over 20 megaflops for some fast Fourier transforms).

CONCLUSION

Advantages of TCOL as an Intermediate Representation

TCOL by itself does not assume any abstract machine model. It is sufficiently extensible to accurately model all of the instructions and operand access modes available on machines with reasonable architectures. Accurately modeling the available target machine resources allows better exploitation of these resources.

With adequate support tools, the use of TCOL as an intermediate representation can significantly speed up and facilitate the construction of compilers, especially those with optimization goals. TCOL suits the multipass nature of an optimizing compiler. Since the program is internally represented by pointers and nodes, traversing the whole program can be done very efficiently any number of times. It also facilitates the implementation of optimizing transformations by simple pointer manipulations.

Disadvantages of TCOL as an Intermediate Representation

Using TCOL as an intermediate representation requires the development of tools. In the absence of adequate tool support, the use of TCOL can complicate, rather than simplify, the implementation of a compiler and it may even introduce an unacceptable compilation overhead.

Tree matching is not as well understood as string matching. Tree matching is often implemented by ad-hoc methods whereas string matching can be implemented by very well understood bottom-up parsing techniques.

Retargetability

We have designed retargetability into the structure of our compiler back-end. Although a large part of our design provided for retargetability, our implementation did not. The prototyping nature of our project allowed us to make many machine-dependent shortcuts in our implementation work. Thus, about 3100 lines (43%) of the total 12000 lines of Pascal code in our compiler back-end is target machine specific. The alert reader may have noticed that the storage-binding expansion phase is heavily machine-dependent and should have been implemented as another table-driven maximal muncher.

We believe that the amount of target-machine-dependent code in a vector compiler back-end can be dramatically reduced in a more careful implementation. The only parts of the back-end code that may be inherently machine-dependent are the routines which implement the code generation directives.

ACKNOWLEDGEMENTS

I thank Rex L. Page for giving me the opportunity to work on this project, and for reviewing an earlier draft of this paper.

REFERENCES


