Compiling C for Vectorization, Parallelization, and Inline Expansion

Randy Allen
Steve Johnson

Ardent Computer Corporation
880 West Maude Avenue
Sunnyvale, California 94086

Abstract

Practical implementations of real languages are often an excellent way of testing the applicability of theoretical principles. Many stresses and strains arise from fitting practicalities, such as performance and standard compatibility, to theoretical models and methods. These stresses and strains are valuable sources of new research and insight, as well as an oft-needed check on the egos of theoreticians.

Two fertile areas that are often explored by implementations are

1. Places where tractable models fail to match practice. This can lead to new models, and may also affect practice (e.g, the average programming language has become more context free over the last several decades).
2. Places where existing algorithms fail to deal with practical problems effectively, frequently because the problems are large in some dimension that has not been much explored.

The present paper discusses the application of a much studied body of algorithms and techniques [Alle 83, KKLW 80, Bane 76, Wolf 78, Wolf 82, Kenn 80, Lamp 74, Huso 82] for vectorizing and optimizing Fortran to the problem of vectorizing and optimizing C. In the course of this work some algorithms were discarded, others invented, and many were tuned and modified. The experience gave us insight into the strengths and weaknesses of the current theory, as well as into the strong and weak points of C on vector/parallel machines. This paper attempts to communicate some of those insights.

1. Introduction

The programming language C has often been called a "typed assembly language," because the statements and the operators in the language are very close to typical machine architectures. For "standard" scalar architectures, a knowledgable programmer can produce close-to-optimal C programs by careful coding and by using well-known coding practices (e.g. using register pointer variables). With vector and parallel architectures, however, this is no longer true. C does not contain any explicit parallel or vector operators; even if it did, it is hard to see how such operators could be used effectively across the wide variety of parallel and vector architectures in existence. As a result, more of the optimizing burden is placed on the compiler for these architectures.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/ or specific permission.

© 1988 ACM 0-89791-269-1/88/0006/0241 $1.50

Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation
Atlanta, Georgia, June 22–24, 1988

Unfortunately, optimizing C is difficult when vector and parallel architectures are involved. To summarize briefly just a few of the problems involved:

1. C programs rarely use matrices, and those that are used are often obtained from dynamic memory and addressed using pointer variables rather than explicit subscripts. As a result, determining when two statements reference independent sections of memory (a determination critical to the success of vectorization and parallelization) is extremely difficult.

2. Embedded assignments, operators with side effects (e.g. ++), and conditional operators (e.g. && and ?:) map very naturally and efficiently to sequential hardware. However, the side effects associated with these operators aggravate the information-gathering process necessary to optimize a program. Additionally, these operations *are* inherently sequential; successful vectorization requires removal or transformation of these operators.

3. The *for* statement in C is considerably less constrained than the Fortran DO loop, making vectorization considerably more difficult. The loop bounds and increment may change within the loop; branches can legally enter loops; and *for* loops almost universally contain operators with side effects to increment the loop variable (if there even is a loop variable).

4. C encourages the use of small functions. Function calls generally inhibit vectorization of any loop containing them. Additionally, the calls hide information necessary for optimization.

5. Unlike ANSI Fortran, C imposes no constraints on subroutine argument aliasing.

6. Because C is commonly used for low-level operating system code, it contains a number of constructs that are difficult to optimize. For instance, ANSI C contains the notion of a *volatile* variable to represent quantities such as status registers whose values may change outside of the context of the program. To give a concrete example, the following loop

```
keyboard_status = 0;
while(!keyboard_status);
```

appears as though it will loop forever. If keyboard_status is declared *volatile*, this loop represents a legitimate (and common) program fragment. Obviously, variables that are declared *volatile* need special treatment at almost every phase of code generation and optimization.

7. The address operator (&) permits modification of variables in subtle ways, thereby increasing the analysis necessary for vectorization and optimization.

These problems are inherent to C and are very difficult to handle in general. However, a large percentage of the problems are directly attributable to the number of function calls, and can be removed by *inlining* procedure calls judiciously[AllC 72]. Some of the barriers removed include:

1. The hidden effects of procedure calls. When the body of the called procedure is revealed in the calling procedure, the optimizer can work based on the real effects of the procedure, rather than assuming the worst case.

2. The problems of argument aliasing become less onerous, and many pointer problems are removed.

3. Since procedure calls cannot in general be executed in vector, inlining procedure calls contained in loops may increase opportunities for vectorization.

This paper describes the implementation of a C compiler focused on the generation of high-quality code for a multi-processor vector machine. For the reasons given above, it was felt that inlining would be an essential part of this compiler. The desire for vectorization, parallelization, and inlining led to a somewhat unusual compiler for C; similarly, the peculiarities of C led to some new strategies in vectorization and parallelization. Section 2 describes the strategy of compilation, detailing the goals we had for the compiler. Section 3 discusses the intermediate form used to represent C programs and the reasons for its choice. Section 4 details the unique features of the C front end that were required to generate the intermediate language. Section 5 describes the processing necessary to make C programs amenable to vectorization; section 6 shows how the same analysis benefits non-vector codes. The implementation of inlining is outlined in section 7, while section 8 will describe special optimizations that become extremely important in the presence of inlining. The conclusion presents an example that illustrates not only the difficulty of compiling C for advanced architectures, but also some of the new insights gained in vectorization by compiling C.

2. Strategy of Compilation

The C compiler described in this paper was targeted for the Titan, a machine whose architecture provides a number of challenges and opportunities for the compiler. Briefly summarized, the Titan is a multiple vector processor machine. One processor on a Titan is comprised of

1. A high speed RISC integer processor.

2. A highly pipelined floating point unit. This unit performs all scalar floating point instructions and all vector instructions (both integer and floating point).

3. A very large vector register file, which serves as a source of operands and targets for the floating point unit. The vector registers can be addressed starting at any location using any vector length and any stride, so that the register set can be viewed (in the extremes) as a collection of 8196 scalar registers, or as four vector registers of length 2048.

A Titan can consist of up to four processors, all of which are connected to a shared memory by a high speed bus. While there are a number of mechanisms that processors can use to communicate, the primary communication/synchronization mechanism is through the shared memory.

In this architecture, the most fruitful optimizations the compiler can perform are:

1. Vectorizing statements in loops to fully utilize the floating point unit. While the segmented nature of the floating unit permits overlap of scalar operations, in practice vector instructions are necessary to keep the pipeline full.

2. Applying multiple processors to a program. Spreading loop iterations among multiple processors can provide significant speedups in many programs.

3. Exploiting low level parallelism. Since the integer and floating point units are completely separate, it is possible to overlap integer and floating point instructions. Similarly, the path to memory from each processor is highly pipelined. As a result, changing the instruction order so that integer and floating point instructions overlap and so that memory access and computation overlap can provide a significant speedup in many programs.

To summarize, the key to efficient execution on the Titan is utilizing parallelism and minimizing memory overhead. Memory overhead can be minimized in two ways: doing good register allocation and "regularizing" memory accesses by using vector operations.

Procedure calls disrupt both vectorization and register allocation. The Titan is intended to be a computation-intensive engine with high quality graphics, so programs running on the machine need frequent access to math libraries and graphics libraries. As a result, the compiler needs an effective strategy for handling procedure calls efficiently. The approach we chose was to allow automatic inlining of procedure calls, with a focus on making the inlining process easy for libraries. Designing this capability into the compiler from the beginning also provided a number of other benefits regarding the size programs that could be compiled and the way in which debugging is accomplished.

3. Choice of C IL

Because C is such a low level language, and because it is often used in contexts where optimization is not important (and often even undesirable!), low level intermediate languages have been the norm for C compilers [GibM 86, John 78]. These intermediate languages are typically very close to a sequential machine architecture, allowing an almost direct mapping from the intermediate representation to machine instructions. Low level representations have also been common in Fortran compilers and vectorizing compilers [Lamp 74, Scar 86]. In fact, many Fortran compilers have translated source programs directly into the PCC intermediate representation—in effect, translating Fortran into C.

For a number of reasons, we departed from the traditional low level representation.

1. Primarily, we did not wish to commit to an instruction sequence too quickly. Normally, C compilers save compile time by lowering the representation immediately. The loss in execution time is small on most scalar architectures, because little global scope is necessary to determine nearly

optimal instruction sequences. With vector and parallel instruction sets, the necessary scope increases dramatically; large sections of code must be analyzed before one statement can be scheduled in vector or in parallel. As a result, the penalty for lowering the representation too quickly can be great.

2. Low level representations greatly increase the difficulty of ferreting out important optimization information. For instance, loops are usually translated directly into goto's and subscripts are mapped directly into pointers. A vectorizer lives or dies by its ability to analyze loops and subscripts; an explicit representation eases the task of vectorization immensely.

3. With a low level representation, *volatile* information is usually lost or hidden. An optimizer cannot work on such a representation unless it can be assured that it is not incorrectly optimizing volatile variables.

4. Given many of the optimizations we anticipated, a high level representation appeared more efficient in terms of compile time. When inlining is applied, many optimizations that are useless in real programs (unreachable code elimination, for instance) are suddenly very important. Dead code is common, and constant propagation is essential (and often creates more dead or unreachable code!). These optimizations are more efficient when invoked on a high level representation, since one node may represent many machine instructions.

4. Implementation of C Front End

The front end of our compiler is based on the PCC2 compiler distributed by AT&T, with a number of ANSI C features (such as function prototypes) added. The common code between our C and Fortran environments includes not only a vectorizer, optimizer, and code generator, but also the type system, a set of routines for building and investigating the IL data structures, and a set of symbol table management routines.

A major change arose in the treatment of C expressions. A C expression is a very potent entity; many side effects are possible, including changes in the flow of control. The vectorizer was vastly simplified by forcing all operations that caused the value of a memory location to change to be made explicit as statements. Thus, the intermediate language has an assignment *statement*, but no assignment *operator*; the C operators ?, :, &&, and || are not representable in expressions.

Consequently, the C front end represents C expressions as a pair: the first element is a sequence of IL statements, typically assignments, while the second element is an IL expression. All of the operators had to be recast to take expressions of this form. For example,

$$(SL_1, E_1) + (SL_2, E_2) \Rightarrow (SL_1 ; SL_2, E_1 + E_2)$$

Assignments are more interesting:

$$(SL_1, E_1) = (SL_2, E_2) \Rightarrow (SL_1 ; SL_2 ; E_1 = E_2, E_1)$$

is the basic idea, but the actual implementation is much more subtle. For example, this transformation, when applied to

$$(SL_1, E_1) = (SL_2, E_2) = (SL_3, E_3)$$

yields (since = associates to the right)

$$(SL_1 ; SL_2 ; SL_3 ; E_2 = E_3 ; E_1 = E_2, E_1)$$

which looks reasonable. However, if E_2 has a side effect (for example, a function call or a reference to a *volatile* variable) then E_2 is in fact invoked twice, which is illegal.

A partial way out of this dilemma is offered by a seemingly unrelated feature of the compiling system—global register allocation. This makes it possible to generate temporary variables with a fair amount of impunity, and to expect that most will be allocated to the same register as their generation, so no additional cost will be incurred. Thus, the actual compilation of assignments looks more like

$$(SL_1, E_1) = (SL_2, E_2) \Rightarrow (SL_1 ; SL_2 ; t{=}E_2 ; E_1{=}t, t)$$

where t is a temporary of the same type as E_1.

This implementation has turned up numerous semantic problems in the Draft ANSI C Standard (ANSI 86), most notably in the notion of *volatile*. If v is a *volatile* variable, and a and b are ordinary variables of the same type, then the ANSI standard does not specify the semantics of

```
a = v = b;
```

Using the transformation described here, the effect is that v is written once, and never read. The ambiguities that arise are sufficiently numerous and painful to call the whole notion of *volatile* into question.

C expressions used in conditional contexts also need great care. For example,

```
while( (SL,E) )
        SSSS
```

must be rewritten as

```
SL;
while( E )
{
    SSSS;
    SL;
}
```

In this case, the list of statements is duplicated. Since this list is in practice, very short, the cost is low. Also, since this list is almost entirely assignment statements, they too tend to be optimized away. In fact, the contexts for the two appearances of *SL* are sufficiently different that they may experience totally different forms of optimization.

5. Processing C for Vectorization

5.1. Overview

Effectively vectorizing C requires a number of enhancements not normally present in vectorizing Fortran compilers. While there are a number of subtle enhancements throughout the compiler, there are two major changes that are worthy of discussion:

1. Special analysis to convert *while* loops into Fortran DO loops. Since C *for* loops are converted to *while* loops by the front end, this transformation is essential to success. While the conversion of *while* loops to iterative loops may seem straightforward, there are a surprising number of intricacies involved in the process.

2. Significant enhancement of induction variable substitution. Standard C programming practices encourage a style of programming in which many auxiliary induction variables are associated with a loop, and the variables are quite often in a form that cannot be easily handled. Uncovering and removing these variables efficiently required a compromise between theoretical elegance and pragmatic coding.

The following sections discuss these conversions in more detail.

5.2. Conversion of While Loops

The C *for* loop is one of many C entities which are easily mapped into good instruction sequences on serial architectures, but which are difficult to treat well on vector and parallel architectures. The *for* loop is relatively unconstrained: the termination and increment conditions can have unusual side effects (particularly if volatile variables are involved); bounds and strides can vary during loop execution (if there even are explicit bounds and strides); and branches can enter and leave the loop at will. Given this lack of regularity, we saw little value in having an explicit representation of *for* loops in our intermediate form. Instead, the C front end represents *for* loops as *while* loops—something it can do straightforwardly without sophisticated analysis.

Of course, *while* loops are not generally as well optimized as Fortran DO loops are, and since many *for* loops are DO loops cast in a different guise, converting *while* loops to DO loops is an important transformation for the compiler. Given the ways in which *for* loops can differ from DO loops, a significant amount of information is necessary to effect the conversion:

1. In order to determine whether branches are entering the loop, control flow information is necessary. The most efficient way to evaluate this condition is using the control flow graph built for scalar analysis.

2. Variation of bounds and strides within the loop is most easily determined by examining the use-def chains built during scalar analysis.

Both of these facts argue that conversion of while loops should occur during scalar optimization. Moreover, since the primary motivation is to utilize vector hardware for *for* loops where possible, the conversion of while loops must come early in the scalar optimization process. If *for* loops are to reap the full benefit of the analysis applied to DO loops, then they must be converted before all the phases which simplify DO loops—induction variable substitution, constant propagation, and dead code elimination. Given these restrictions, the proper place to convert *while* loops is immediately after use-def chains have been constructed.

This placement is not without problems. In particular, converting a *while* loop to a DO loop changes the use-def chains, thereby requiring the transformation to incrementally reconstruct the chains that correspond to the transformed code. This is not an entirely straightforward task, as the following simple example illustrates:

```
i = n;                      i = n;
while(i) {                  DO dummy = n, 1, -s
  ...                         ...
  temp = i;                   temp = i;
  i = temp - s;               i = temp - s;
}                           ENDDO
```

When the while loop is transformed, the updated chains must reflect proper definition points for the variables *n* and *s* used in the DO loop conditions. In this example, the updating is not difficult, and simply involves a transitive transfer from the locations identified as the sources for the initial value and the increment value. In general, the problem is much more difficult, particularly because graphics code is one of the main benefactors of vectorization. Graphics code typically transforms 4x4 matrices with operations that can be beneficially vectorized. Knowing that the vector length in such loops is small enough that a strip loop is not required is very important and is often difficult, depending on the exact way in which the *for* loop is coded.

As an aside, the Titan compiler also has plans for optimizing *while* loops that are truly *while* loops, and not merely iterative loops lurking in disguise. A prime example of such a loop is code that operates on a linked list. Such a loop cannot be vectorized with any benefit, but it can be spread across multiple processors by pulling the code for moving to the next element into the serialized portion of the parallel loop. There are also a number of cases in which the condition of a loop is necessary only to compute the termination point. In such cases, computing the termination criteria can often be pulled into a separate loop. The resulting bound can then be used in iterative loops representing the major portion of the computation, which can then be vectorized [AllK 85].

5.3. Induction Variable Substitution

Induction variable substitution takes code that has been strength-reduced by hand and makes it more amenable for vector hardware. For instance, the loop on the left cannot be directly executed in vector:

```
IV = N
DO I = 1, N                 DO I = 1, N
  A(IV)=A(IV)+B(I)            A(N-I+1)=A(N-I+1)+B(I)
  IV = IV - 1
ENDDO                       ENDDO
```

The reason is that the way in which A varies with the DO loop index is implicit, rather than explicit. Induction variable substitution applied to this loop will produce the loop on the right, where the variance is explicit and vectorization is trivial.

In Fortran, induction variable substitution is useful primarily because many users have programming habits based on Fortran 66 (which limited the forms that subscripts and loops could take) and because many users formed the habit of strength reducing subscripts by hand to substitute for poor optimizing compilers. In this context, relatively straightforward schemes are able to uncover virtually all of the auxiliary induction variables that appear in practice [Wolf 78, AllK 81]. However, C presents a more challenging situation.

Because the C front end does not do a sophisticated analysis for side effects on expressions that appear with ++ expressions, the intermediate code it generates is ripe with opportunities for induction variable substitution. To give a simple example, source code such as

```
while(n) {
  *a++ = *b++;
  n—;
}
```

is translated by the C front end into the following:

```
while(n) {
  temp_1 = a;
```

```
    a = temp_1 + 4;
    temp_2 = b;
    b = temp_2 + 4;
    *temp_1 = *temp_2;
    temp_3 = n;
    n = temp_3 - 1;
}
```

This loop can be straightforwardly vectorized (it is, after all, only a vector copy) once all the garbage is cleared away. Before it can be vectorized, the key assignment must be converted into a form similar to

```
*(a + 4*i) = *(b + 4*i)
```

where i is assumed to be the DO loop induction variable. The problem is that this form can be created only by substituting the assignment to temp_1 and temp_2 forward into the star assignment. This substitution cannot be correctly made, however, until the updates to a and b are moved forward. As a result, straightforward techniques cannot handle this loop.

There is a clear theoretical solution to this problem. Straightforward extensions of Morel and Renvoise's work on partial redundancies can be used to uncover and remove general classes of induction variables [MoRe 79, Chow 83, Lich 87]. For many pragmatic reasons, we chose not to utilize the theoretical solution:

1. Foremost was space considerations. Scalar dataflow information commonly eats up large amounts of memory; were we to take this approach, we would have to compute extra dataflow attributes solely for induction variable substitution. We felt that this might require more memory than we could give it.

2. Secondary was graph reconstruction. Induction variable substitution is only one of many transformations that are driven off scalar dataflow information. By necessity, it must occur early in the optimization process; as a result, dataflow information must be updated to reflect the changes imparted by the substitution. We knew we could accomplish this incrementally (and thereby efficiently) with standard techniques. We were not certain that we could incrementally update the information based on partial redundancies, and we felt that reconstructing the graph from scratch would be far too expensive.

3. Finally, we were able to derive a heuristic solution that is effective and efficient on codes encountered in practice. In the worst case, this solution is extremely inefficient, requiring n passes over a loop (where *n* is the number of statements in the loop). However, in practice we have never seen this behavior; the average case requires the same simple pass over the loop that is needed in the straightforward algorithm.

The heuristic solution employed in the Titan compiler is fairly simple: when a statement is rejected for substitution only because a later statement redefines a variable used by that statement, the later statement is marked as "blocking" the first statement. When a blocking statement is substituted forward, all the statements it blocks are reexamined for substitution. In this approach, backtracking is never done unless it is guaranteed to give some substitution. Furthermore, most of the analysis necessary to substitute the statement need not be repeated. As a result, the backtracking is rarely invoked, and is extremely efficient when it is invoked.

In the example above, backtracking substitutes the assignments to temp_1 and temp_2 after the assignments to a and b have been substituted. As a result, the loop is converted to a form that is trivially vectorized.

6. Dependence-Driven Optimizations

There are probably far more C programs that do not vectorize than do, even when the vectorizer is specifically tuned for C. However, the dependence graph used in vectorization can be used to optimize in other contexts. Some of the ways in which the Titan compiler uses its dependence graph to optimize "non-vector" programs include:

1. Register allocation. The dependence graph used in vectorization has a dual nature that permits it to be an effective basis for register allocation as well as for vectorizing. A data dependence between two statements implies that their execution order must be fixed in some way. When restructuring a program, the desire is to have as few dependences as possible, to provide the maximum freedom for reordering execution. The duality arises from the fact that two statements can be data dependent only if they access a common memory location. As a result, data dependences pinpoint the memory locations that are most frequently accessed, providing a mechanism for exploiting the memory hierarchy of the Titan.

2. Instruction scheduling. The array dependence graph accurately indicates all the execution constraints involving array references. This information permits far more levity in instruction scheduling, since most array references are usually independent and instruction scheduling normally occurs at a level where this fact is difficult to determine. Information from the dependence graph is passed back to the code generation to allow better overlap of integer and floating point computations, and also to allow better overlap of memory access and computation.

3. Strength reduction. Because classic vectorizing transformations such as induction variable substitution deoptimize programs that do not vectorize, strength reduction is a very important optimization in the Ardent compiler. Our algorithm is unique in that it utilizes the array dependence graph to simultaneously reduce expensive operations, remove loop invariant expressions, and eliminate common subexpressions. The reduction algorithm must be very careful of the parallelism present in a program, since strength reduced operations are by their very nature sequential.

Details of these optimizations are available elsewhere [AllL 88]. To illustrate their power, consider the following sample loop:

```
p = &x[1];
q = &x[0];
for(i=0; i<n-2; i++)
   p[i] = z[i] * (y[i] - q[i]);
```

This loop, which is a typical loop used in backsolving linear systems, cannot be correctly run in vector or parallel because the array q uses values stored into p on previous iterations. This use is quite regular, however; the Titan vectorizer is able to recognize this regularity and pull the values up into registers. By doing so, it eliminates some memory access constraints on instruction scheduling, thereby allowing the instruction scheduler to completely overlap the integer and floating point instructions in the loop and also the stores of x to memory with the computation. Finally, strength reduction is able to eliminate all the integer multiplications within the loop. The resulting intermediate code is:

```
f_reg1 = x[0]
temp_z = &z;
temp_y = &y;
temp_x = &x + 4;
for(i=0; i<n-2; i++) {
  f_reg1 = *temp_z * (*temp_y - f_reg1)
  *temp_x = f_reg1;
  temp_x = temp_x + 4;
  temp_y = temp_y + 4;
  temp_z = temp_z + 4;
}
```

When the original loop is compiled with only scalar optimization on the Titan, it executes at 0.5 megaflops. When the vectorization information is used to produce the second form, the execution rate is 1.9 megaflops, which is within 5% of the best possible code for this loop.

7. Implementation of Inlining

As stated previously, there were motivations for inlining in the Titan C compiler:

1. Efficient inlining of small static functions common in C.

2. Effective code generation in the presence of calls to math (or other) libraries. For instance, it was very desirable that a C call to a low level linear algebra routine (such as DAXPY) generate the fastest code possible, since a user who writes such a call is probably expecting good performance.

The first goal is not very difficult, because all the information necessary for inlining and compilation is available in one file at one time. A number of compilers are able to inline functions of this form. The second goal is more difficult, however, and requires careful design throughout the compiler.

In order to inline functions from other files, the intermediate representation for functions must be saved in an easily accessible form. To permit this, we eliminated all hard pointers from the IL. This allowed us to to "page" tables if necessary, to compile very large files, and to save the parsed form of procedures in catalogs. As a result, math libraries can be "compiled" into databases and used as a base for inlining, much as include directories are used as a source for header files.

Using databases of parsed procedures requires significant information gathering and some program transformations. For instance, static variables inside a procedure stored in a database must be made externally known, so that values are correctly maintained regardless of whether the procedure is called normally or through inlining. Because external variables are not optimized as well as other variables, this makes the detection of static variables that can be safely moved to automatic storage an important optimization. Also, since C permits recursion (which can lead to infinite inlining if care is not taken), and since inlined functions may inline other functions, order is very important.

While the use of relocatable data structures caused some problems throughout the compiler, it had a number of unexpected benefits. The debugger, for instance, works directly on the same symbol table that the compiler builds during compilation.

8. Special Inlining Optimizations

Many optimizing techniques are virtually useless to invoke on code written by programmers, because intelligently written code will never contain any opportunities for their use. For instance, code written by a programmer should never contain unreachable code; if it does, it is usually an error (and flagged as such by many compilers). When inlining is invoked, this situation changes dramatically. Inlining tailors a procedure designed to handle many cases to a specific invocation; as a result, large amounts of dead and unreachable code result.

Not only does inlining shift the focus of optimizations, it affects the order in which optimizations are done. Consider, for instance, detection of unreachable code. Typically, unreachable code is detected at the time basic blocks are constructed; if a block has no predecessors, it is unreachable. This approach is not effective with inlined procedures, however, because constant propagation is necessary to realize that code is unreachable. Consider, for instance, the following example.

```
daxpy(*x, y, 0.0, z);
...
void daxpy(float *x, float y, float a, float z)
{
  if (a == 0.0)
    return;

  *x = y + a * z;
}
```

When inlined, this becomes

```
  in_x = x;
  in_y = y;
  in_a = 0.0;
  in_z = z;

  if (in_a == 0.0)
    goto lb_1;

  *in_x = in_y + in_a * in_z;
  goto lb_1;

lb_1:;
```

In this somewhat contrived example, the floating point assignment will never be executed, because the first branch to lb_1 will always be taken. This fact cannot be detected until constant propagation reveals that in_a is always equal to 0.0. This example illustrates a general point: the information provided by the specific parameters at a call site permits a large amount of optimization.

While unreachable code does not affect the program speed, it does affect the program space. From a theoretical point of view, there are at least three obvious methods for detecting and eliminating unreachable code:

1. Perform IF conversion [AKPW 83] on the whole program; any statement whose guard is false is unreachable.

2. Rebuild basic blocks after optimization. Any block with no predecessors (with some accommodation for loops) is unreachable.

3. Employ a constant propagation technique specifically designed to accommodate constant propagation with unreachable code elimination, such as Wegman-Zadeck [WegZ 85].

We rejected the first two possibilities on the grounds of efficiency. Not only did both techniques require reanalyzing the entire program, but IF conversion could also introduce some inefficiency into the generated code. After some thought, we also rejected the third approach as being infeasible for our implementation. There were a number of reasons for this decision:

1. We drive a number of optimizations off the use-def graph, not just constant propagation and unreachable code elimination. It was not clear that the modifications necessary to the data structures would permit us to drive these other optimizations.

2. More specifically, induction variable substitution and *while* loop conversion are complicated by the presence of identity assignments. While both transformations can be done in this context, the transformations themselves tend to remove identity assignments, thereby undoing much of the analysis that is actually needed later during constant propagation.

3. In hindsight, the memory requirements of inserting identity assignments at birthpoints would probably have been intolerable for a production compiler. The vectorizer inserts identity assignments at a number of birthpoints in order to determine which variables must be allocated to local memory within parallel loops. Even this limited use of identity assignments tends to increase the number of nodes in the graph (and thereby the graph size) to an unacceptable level. The extent to which this transformation is applied has had to be limited several times due to large memory requirements.

Rather than adopt any of these approaches, we implemented an heuristic solution to the problem which turns out to be extremely effective in practice. During constant propagation, the compiler eliminates code that is detected as unreachable due to if conditions being simplified to false or true, loops which are detected as having zero iterations, etc. When a statement is eliminated as being unreachable, all statements that its definition reaches are added to a list. All constant assignments whose definitions can reach any statement in this list are then added to the heap for another round of possible propagation. This approach tends to pick up almost all constants whose definitions are blocked by unreachable definitions; it does not eliminate all unreachable code that arises in practice. In particular, code immediately following branches that are always taken is difficult to uncover as unreachable during constant propagation. The vectorizer has a separate postpass that is invoked when inlining is enabled to eliminate this kind of unreachable code. The postpass is a quick heuristic and is not as effective as reconstructing basic blocks. On the other hand, it is very effective in practice and requires less compile time.

One other transformation is useful in the context of inlining. Array rows passed by reference into a procedure lead to subscripted references whose base arrays are also subscripted references. Such references are perfectly legitimate, but are virtually impossible to successfully analyze in a vectorizer. The inlining phase contains a special pass to "promote" such references into a standardized form that is easy to recognize and handle.

9. An Example

In order to illustrate the difficulties involved in compiling C for advanced architectures, we present an example. The example also illustrates how the techniques described in this paper work together to compile difficult loops into good code.

```
main()
{
  float a[100], b[100], c[100];
  daxpy(a,b,c,1.0,100);
}

void daxpy(float *x, float*y, float *z,
       float alpha, int n)
{
  if (n <= 0)
    return;
  if (alpha == 0)
    return;

  for(;n; n—)
     *x++ = *y++ + alpha * *z++;
}
```

The routine daxpy is a C analog of the Fortran BLAS routine; it adds a scalar multiple of one vector to another. This coding does differ from the BLAS routine in that it can place the results in a different vector. This is quite obviously a vector function; such a routine coded in Fortran would be trivially compiled as such by a good vectorizing compiler. This C routine cannot be safely vectorized, because C imposes no restrictions on argument aliasing. *x*, *y*, and *z* could be pointers into the same array, invalidating the analogous vector operation.

This routine can be automatically vectorized by adding in a pragma stating that the loop is safe to vectorize or by invoking a compiler option that states that pointer parameters have Fortran semantics—stores into any parameter do not affect values fetched from a different parameter. However, we can also inline *daxpy*, producing the following intermediate representation:

```
main()
{
   float a[100],b[100],c[100];
   float *in_x, *in_y, *in_z, float in_alpha,
       *in_1, *in_2, *in_3, *in_4;
   int in_n;

   in_x = &a;
   in_y = &b;
   in_z = &c;
   in_alpha = 1.0;
   in_n = 100;

   if (in_n <= 0)
     goto lb_1;
   if (in_alpha == 0.0)
     goto lb_1;

   while (in_n)
   {
     in_2 = in_x;
     in_x = in_2 + 4;
     in_3 = in_y;
     in_y = in_3 + 4;
     in_4 = in_z;
     in_z = in_4 + 4;
     *in_2 = *in_3 + in_alpha * *in_4;

     in_1 = in_n;
     in_n = in_1 - 1;
   }
```

```
    lb_1:;
}
```

Although inlining has eliminated the aliasing problem, that fact is masked by the profusion of temporaries that are necessary to preserve the semantics of C (for instance, a variable such as *n* could be a function call). Induction variable substitution and wh*ile* loop conversion simplify the problem considerably, yielding the following code:

```
main()
{
  float a[100],b[100],c[100];
  float *in_x, *in_y, *in_z, float in_alpha;
  int in_n;

  in_x = &a;
  in_y = &b;
  in_z = &c;
  in_alpha = 1.0;
  in_n = 100;

  if (in_n <= 0)
    goto lb_1;
  if (in_alpha == 0.0)
    goto lb_1;

  do fortran temp_i = 0, n-1, 1
  {
    *(in_x + 4*temp_i) = *(in_y + 4*temp_i) +
            in_alpha * *(in_z + 4 * temp_i);
  }
  in_x = in_x + 400;
  in_y = in_y + 400;
  in_z = in_z + 400;
  in_n = in_n - 100;

  lb_1:;
}
```

After constant propagation and dead (not unreachable) code elimination, this becomes:

```
main()
{
  float a[100],b[100],c[100];
  int temp_i;

  do fortran temp_i = 0, 99, 1
  {
    *(&a + 4*temp_i) = *(&b + 4*temp_i) +
              *(&c + 4*temp_i);
  }
}
```

At this point, the code is in a form that the vectorizer can handle. While the implicit representation of subscripts as star operations is not difficult to handle, it did require some special tuning in the vectorizer. In this form, the Titan vectorizer is able to recognize that the references are independent and produce the following vectorized code (the colon notation indicates vector operations and the do parallel indicates that the vector operations may be done in parallel):

```
main()
{
  float a[100],b[100],c[100];
  int vi, vr;

  do parallel vi = 0, 99, 32
  {
    vr = min(99, vi+31);
    a[vi:vr:1] = b[vi:vr:1] + c[vi:vr:1];
  }
}
```

On a two processor Titan, this code executes 12 times faster than the scalar version of the same routine.

10. Current Status and Future Work

At present, the Titan C compiler provides a good balance between compile time and optimization. The compiler has compiled all of UNIX™[*], several benchmarks, and a large graphics package called Doré™[**] (Dynamic Object Rendering Environment). Of these packages, Doré has provided the most interesting experiences. Major pieces of Doré have been compiled with full vectorization and parallelism, and have achieved impressive speedups. The one deficiency which we uncovered in vectorizing Doré was arrays embedded within structures. We originally did not put much effort into handling this kind of construct, since we did not think it would arise very often. Given the prevalence with which this appears within graphics code, our decision was poor. It does not take much additional effort to correctly handle this type of array, although the notion of what exactly is the array becomes a little unsettled.

In the future, we plan to enhance this effort in at least two ways. First, we plan to enhance the parallelization to include list and graph structures, as described earlier in the paper. This enhancement is not that difficult, although it does require an assumption that each motion down a pointer goes to independent storage. Parallelizing this type of code will enable a wider range of programs to utilize the multiple processors in the Titan.

Second, we plan to integrate the inlining and vectorization features into languages similar to C++, where concepts such as *classes* can provide valuable information regarding the safety of many transformations—information which is not always available in C itself.

11. Conclusions

This paper has attempted to convey three lessons. At one level, we have attempted to demonstrate, primarily by example, the careful balance necessary between the theoretical and the practical to build a successful compiler. The lesson that we have tried to convey is that a firm theoretical basis is important for an optimization, but it is also important to recognize that theory almost always bends somewhat to satisfy pragmaticism.

Second, we have tried to illustrate the importance of balancing the work of analyzing a program among various phases of the compiler. The C front end is fairly quick and simple, because it is able to generate straightforward code, secure in the knowledge that good register allocation and scalar optimization will clean up the any glaring mistakes. Similarly, the vectorizer is safe in propagating address constants and in performing induction variable substitution because it knows that strength reduction and subexpression elimination will undo any damage it has done. This philosophy of spreading the analysis among the phases that are most apt to perform it has turned out to be an enormous win in our compiler.

[*] UNIX is a registered trademark of AT&T.
[**] Doré is a trademark of Ardent Computer Corporation.

Finally, we have tried to espouse a different philosophy toward compiling C. C is a language that was designed with the intent of providing a mechanism by which programmers could access features of hardware efficiently from a relatively high level. Because of this natural mapping, compilers for C have tended to immediately map an input program to a representation that is very close to hardware. Our compiler has a different philosophy: representing the program in a form which allows for intelligent analysis before deciding upon a machine level execution. We believe that this is a viable way of compiling C, and that as architectures tend to move more toward parallelism, this approach will produce a better mapping from programs to hardware.

12. References

[AllC 72]
Allen, F.E. and Cocke, J., "A catalogue of optimizing transformations," *Design and Optimization of Compilers*, R. Ruskin, ed., Prentice-Hall Englewood Cliffs, N.J. 1971, 1-30.

[AKPW 83]
Allen, J.R., Kennedy, K., Porterfield, C., and Warren, J., "Conversion of control dependence to data dependence," *Conference Record of the Tenth Annual Symposium on Principles of Programming Languages*, Austin, Tx., Jan. 1983, pp. 177-189.

[Alle 83]
Allen, J.R., "Dependence analysis for subscripted variables and its application to program transformations," PhD dissertation, Department of Mathematical Sciences, Rice University, Houston, Tx., April, 1983.

[AllK 82]
Allen, J.R. and Kennedy, K., "PFC: a program to convert Fortran to parallel form," MASC TR82-6, Department of Mathematical Sciences, Rice University, Houston, Tx., March, 1982. Reprinted in *Supercomputers: Design and Applications*, K. Hwang, editor, IEEE Computer Society Press (1985), pp. 186-205.

[AllK 86]
Allen, J.R. and Kennedy, K., "Programming Environments for Supercomputers," in *Supercomputers: Algorithms, Architectures, and Scientific Computation*, F.A. Matsen and T. Tajima, ed., University of Texas Press (1986), pp. 19-38.

[AllL 88]
Allen, J.R. and Lew, S., "Why Even Scalar Machines Need Vector Compilers,", Technical Report, Ardent Computer, Jan., 1988.

[ANSI 86]
"Draft Proposed American National Standard Programming Language C", X3.159-198x, American National Standard Committee X3J11, October, 1986.

[Bane 76]
Banerjee, U., "Data dependence in ordinary programs," Report 76-837, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, November 1976.

[Chow 83]
Chow, F., "A portable machine-independent global optimizer—design and measurements,", Technical Report 83-254, Dept. of Electrical Engineering and Computer Science, Stanford University, Dec. 1983.

[GibM 76]
Gibbons, P.B. and Muchnick, S.S., "Efficient instruction scheduling for a pipelined architecture," *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, Palo Alto, Ca., June, 1986, pp 11-16.

[Huso 82]
Huson, C.A., "An in-line subroutine expander for Parafrase," M.S. Thesis, University of Illinois at Urbana-Champaign, 1982.

[John 78]
Johnson, S.C. "A portable compiler: theory and practice," *Proceedings of the Fifth Annual Symposium on Principles of Programming Languages*, Tucson, AZ., Jan., 1978, 97-104.

[Kenn 80]
Kennedy, K., "Automatic Translation of Fortran Programs to Vector Form," Tech Report, Department of Computer Science, Rice University, October, 1980.

[KKLP 81]
Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M., "Compiler transformation of dependence graphs," *Conf. Record of the Eighth ACM Symposium on Principles of Programming Languages*, Williamsburg, Va., January 1981.

[KKLW 80]
Kuck, D.J., Kuhn, R.H., Leasure, B., and Wolfe, M., "The Structure of an advanced vectorizer for Pipelined Processors," *Proc. IEEE Computer Society Fourth International Computer Software and Applications Conf.*, IEEE, Chicago, October 1980.

[Lamp 74]
Lamport, L., "The parallel execution of DO loops," *Communications of the ACM*, Vol 17, No. 2, pp. 83-93, February, 1974.

[Leas 76]
Leasure, B.R., "Compiling serial languages for parallel machines," Report-76-805, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, November 1976.

[Lich 86]
Lichnewsky, A., private communication.

[MoRe 79]
Morel, E. and Renvoise, C., "Global optimization by suppression of partial redundancies," *Comm. ACM* 22, 2 (Feb. 1979).

[Scar 86]
Scarborough, R.G. and Kolsky, H.G., "A vectorizing FORTRAN compiler," *IBM Journal of Research and Development*, March, 1986.

[WegZ 85]
Wegman, M., and Zadeck, F.K., "Constant propagation with conditional branches," Twelfth POPL, New Orleans, La., Jan., 1985.

[Wolf 78]
Wolfe, M.J., "Techniques for improving the inherent parallelism in programs," Report 78-929, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, July 1978.

[Wolf 82]
Wolfe, M.J., "Optimizing Supercompilers for Supercomputers," PhD Dissertation, University of Illinois at Urbana-Champaign, October, 1982.