CONVEX C Optimization Guide

Document No. 720-001130-200

First Edition
May 1990

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX C Optimization Guide
Order No. DSW-089
First Edition

Printed in the United States of America

Revision Information for
CONVEX C Optimization Guide

Edition	Document No.	Description
First	720-001130-200	Released with CONVEX C V4.0, April 1990.

Table of Contents

Appendices

Preface

This guide presents a method for optimizing C programs, with principles, directives, compiler options, and analysis. Background information and concepts presented in the first few chapters form a foundation for methods presented later in the book. Examples show the use of command-line options, compiler directives, and various tricks and tips to control and enhance scalar, vector, and parallel optimization.

Producing an efficient program requires efficient algorithms and efficient implementation. The techniques of writing an efficient algorithm are beyond the scope of this guide. The guide assumes you have chosen the best possible algorithm for your problem and helps you get the best possible performance out of that algorithm.

Intended Audience

The *CONVEX C Optimization Guide* is for experienced C programmers using the CONVEX compiler. It is of most interest to those using the full, vectorizing version of the compiler, although users of the scalar optimizing base-level compiler may find some sections to be of interest also. Readers need not be familiar with the CONVEX implementation of scalar, vector, and parallel optimization to use this guide.

Organization

This document consists of the following chapters:

- Chapter 1 introduces the CONVEX approach to program optimization and introduces terms and concepts needed to understand how the CONVEX C compiler exploits the CONVEX C100 and C200 Series architecture.
- Chapter 2 discusses scalar optimization and describes transformations the compiler can perform when you compile your program for scalar optimization (command line options `-no`, `-O0`, and `-O1`).
- Chapter 3 discusses vector optimization and describes transformations the compiler can perform when you compile your program for vector optimization (command line option `-O2`).
- Chapter 4 discusses parallel optimization and describes transformations the compiler can perform when you compile your program for parallel optimization (command line options `-O3`).
- Chapter 5 presents a strategy for developing programs and provides examples of the use and effects of compiler options and directives that affect optimization.
- Chapter 6 discusses programming constructs that can aid or hinder optimization.
- Chapter 7 discusses the special optimization problems caused by aliasing, which usually arises from the use of pointers.

- Chapter 8 discusses some of the tricks and tips used to optimize programs.
- Chapter 9 discusses common problems related to optimization and presents some possible solutions.

Related Reading

- For more information on the compiler, refer to the *CONVEX C User's Guide* and *CONVEX C Language Reference Manual.*
- For more information on `CXpa`, the performance analyzer, refer to the *CONVEX Performance Analyzer (CXpa) User's Guide.*
- For more information on `csd`, the source-level debugger, refer to the *CONVEX Consultant User's Guide,* 8th edition.
- For more information on parallel processing on CONVEX supercomputers, refer to the Fall 1988 issue of *Vector* (Volume II, Number 3).
- For information on parallel programming in assembly language, refer to the *CONVEX Assembly-Language User's Guide* and the *CONVEX Architecture Reference.*
- For a complete discussion of data dependence, refer to "Advanced Compiler Optimizations for Supercomputers," by David A. Padua and Michael J. Wolfe, in the December 1986 issue of *Communications of the ACM* (Volume 29, Number 12).
- For more information on scalar optimizations, refer to *Crafting a Compiler,* by Charles Fischer and Richard LeBlanc, Jr., Benjamin/Cummings Publishing Company, Inc., 1988.

For more information on the C language, refer to the following books:

— *American National Standard for Information Systems -- Programming Language C.* Document Number: X3J11/90-013.

— *C: A Reference Manual,* by Samuel P. Harbison and Guy L. Steele, Jr., Prentice-Hall, Inc., 1987.

— *The C Programming Language,* by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1988.

Notational Conventions

The following conventions are used in this document:

- *Italics* is used for book and periodical titles, for emphasis, or to identify an important word or concept that is being defined or used for the first time.
- `Monospace` is used for examples. It is also used within text to indicate compiler options and directives, command names, file names, keywords, and variables.
- **Boldface** is used in screen examples indicates user input.
- The special notation [`first:last`] for array subscripts means all elements of the array from `first` to `last`. This is not a valid subscript notation in CONVEX C, but appears in pseudocode examples as a shorthand form for vectors.

Chapter 1
The Basics

Optimization transforms program code to return correct answers in a shorter time than before. To optimize your program, the CONVEX C compiler:

- Eliminates unnecessary operations.
- Arranges operations in the most efficient order.
- Replaces slow operations with faster equivalents.
- Fully exploits the features of the CONVEX architecture.

The CONVEX C compiler offers five optimization levels. The transformations performed depend on the optimization level you select.

At the lowest level (`-no`) no higher-level optimizations are performed. The only optimizations performed at `-no` are machine-dependent scalar optimizations, which fully exploit the CONVEX C Series computer architecture. `-no` is the default optimization level. These machine-dependent optimizations take place at the machine-instruction level and cannot be disabled.

As the optimization level increases, new optimizations are added. Going to `-O0` adds machine-independent scalar optimization. These machine-independent optimizations are elaborated at `-O1`. Machine-dependent optimizations are performed as well.

`-O1` is the highest scalar optimization level supported by the CONVEX C compiler. The compiler flag `-O` (no numeral) is a synonym for `-O1`.

Increasing the optimization level to `-O2` adds *vector* optimization. Increasing the optimization level to `-O3` adds vector and *parallel* optimization. Scalar optimizations are still performed as well.

Vector and parallel optimizations are not performed by the scalar optimizing version of the CONVEX C compiler bundled with each machine. Optimization levels `-O2` and `-O3` do not exist on this compiler. To generate parallel or vector code, you must have access to the full vectorizing version of CONVEX C, which is an optional product available through your CONVEX sales representative. In addition, to execute parallel code, you must have a system with multiple CPUs.

This document deals primarily with vector and parallel optimization and is intended for users of the full vectorizing C compiler. Users of the scalar optimizing compiler will find the material on scalar optimizations presented in Chapter 2 to be of interest. Users of the full vectorizing C compiler must also understand scalar optimizations, which continue to affect optimization efforts at higher optimization levels.

Machine-Independent Scalar Optimization

Machine-independent scalar optimizations are performed at two levels:

- Local (basic-block).
- Global (function).

A basic block is a linear sequence of statements that ends with a conditional or unconditional branch. There can be no branches within the body of a basic block. At optimization level `-O0`,

the compiler looks at only one basic block at a time when performing machine-independent optimizations (optimization is local to a basic block). At optimization level `-O1`, the compiler looks at multiple basic blocks within a function (optimization is global to a basic block). Optimization is always local to a function.

Basic blocks should not be confused with the block or compound statement defined by C.

To improve performance, machine-independent optimizations:

- Minimize the number of memory accesses.
- Simplify expressions.
- Eliminate redundant operations.
- Replace variables with constants.
- Replace slow operations with faster equivalents.

Vector Optimization

The use of vector instructions (vectorization) can greatly improve the performance of programs that manipulate arrays. If a loop adds corresponding elements of two arrays, for example, vector registers and instructions can perform simultaneous additions on up to 128 elements of each array.

Vector optimization does more than just vectorize loops. Many times, a loop cannot be vectorized as written. The CONVEX C compiler performs a number of transformations to greatly increase the number of loops that can be vectorized. In an application that relies heavily on manipulation of arrays, vector optimization can reduce execution time by up to 90 percent.

When the `-O2` option is specified on the `cc` command line, the compiler performs vector optimization. The compiler performs scalar optimization on loops that cannot be vectorized.

Parallel Optimization

Scalar and vector optimization reduce a program's CPU time. Parallel optimization increases CPU time, but decreases turnaround time by spreading the work across multiple CPUs. The actual savings in turnaround time depends greatly on the application that is being compiled.

When the `-O3` option is specified on the `cc` command line, the compiler performs both parallel and vector optimization. Scalar optimization is performed on loops that cannot be parallelized or vectorized.

On the CONVEX C200 Series machine, each CPU automatically seeks the next available piece of work and executes it as soon as possible. This feature, known as ASAP (Automatic Self-Allocating Processors), eliminates processor idle time and optimizes system use in a multiuser environment. ASAP allows a program to obtain CPU cycles as they become available, without the overhead of a system call.

A sequence of instructions that can execute on a single CPU is called a *thread*. Every program has at least one thread. Parallel programs have multiple threads that are divided among available processors.

At `-O3`, programs are automatically parallelized the same way they are vectorized—at the loop level. The compiler divides loop iterations into groups that can be placed on separate threads and generates code that does not depend on the number of available CPUs. You can parallelize constructs that are not loops using tasking directives.

At best, parallelization can improve turnaround time by a factor of N, where N is the number of available CPUs on your system. Limitations imposed by algorithms prevent most programs from realizing more than a fraction of this theoretical improvement. The information contained in this guide can help you get the best possible parallel performance out of your algorithm.

Optimization Tools

Debugging code that the compiler had modified can be very difficult. Code must be tested and debugged before optimization begins. The `csd` source-level debugger is a tool developed for this purpose. Part of the CONVEX Consultant package, `csd` can set process breakpoints, examine registers, and display backtraces of the stack. For information on how to use `csd`, see the *CONVEX Consultant User's Guide.*

The CONVEX Performance Analyzer, CXpa, lets you examine your program's performance at the routine, loop, and basic-block level. You can use CXpa, or one of the older profilers contained in the CONVEX Consultant, to track effects of optimizations. For information on how to use CXpa, see the *CONVEX Performance Analyzer (CXpa) User's Guide.*

Conclusion

The CONVEX C compiler performs optimization at several levels. Each higher level adds new optimizations to those performed at lower levels. At higher levels, the compiler weighs the benefits of each optimization to produce the fastest possible code without endangering the accuracy of calculations. CXpa and `csd` are tools to aid you in understanding your program and eliminating logic errors.

Chapter 2
Scalar Optimization

The CONVEX C V4.0 compiler performs two types of scalar optimizations on your code:

- Machine-dependent optimizations.
- Machine-independent optimizations.

Machine-dependent optimizations exploit the special features of the CONVEX C Series computer architecture. At the lowest optimization level (`-no`), only machine-dependent optimizations are performed. These optimizations take place at the machine-instruction level and cannot be disabled.

Machine-independent optimization begins at optimization level `-O0`. This optimization begins at the level of a basic block. The basic block is a linear sequence of statements with a single entry and a single exit. It ends with a conditional or unconditional branch to another block. There are no conditional or unconditional branches within the body of a basic block.

At level `-O0`, the compiler looks at one basic block at a time when performing machine-independent optimization. Machine-independent optimization is local to a basic block. At optimization level `-O1`, the compiler looks at multiple blocks with a single function. Machine-independent optimization is global to a basic block, but still local to a function.

Note

The assembly-language listing produced by the compiler's `-S` option shows the basic blocks in the final, optimized assembly code. At optimization level `-no` there is a one-to-one correspondence between these basic blocks and the basic blocks in the original C code. At higher optimization levels, this may not always be the case. If a basic block is dead code, such as an unreachable alternative in an `if` statement, it can be eliminated by the compiler at higher optimization levels. The number of basic blocks shown in the assembly-language output (or by block-level profilers `bprof` and CXpa) typically decreases as the optimization level is increased.

Basic blocks should not be confused with the block or compound statement defined by C.

The following sections describe transformations performed when you compile a program for scalar optimization (that is, at optimization levels `-no`, `-O0`, and `-O1`). Most scalar optimizations are performed automatically, without any need for programmer intervention. Programmers who are unfamiliar with the optimizations performed by the compiler often waste valuable time duplicating these optimizations by hand and may introduce logic errors in the process. For this reason, it is strongly suggested that readers not skip this chapter.

Optimizations Performed at -no

At optimization level -no, the compiler performs machine-dependent optimizations. These optimizations create object code that exploits the scalar features of the CONVEX architecture.

Instruction Scheduling

Each CPU on a CONVEX supercomputer has multiple *functional units.* On a C200 Series processor, for example, add, divide, and store operations are performed by three separate functional units. As a result, an add, divide, and store can take place concurrently. Instruction scheduling rearranges machine instructions to use the functional units most effectively (that is, to achieve the highest level of concurrency). At optimization level -no, instruction scheduling is performed on the instructions derived from a single higher-level language source statement.

Compare two assembly codes for the following statement:

```
a= (b+ c*d) / e-f;
```

Original Code		Optimized Code	
ld.w	d,s0	ld.w	d,s0
ld.w	c,s1	ld.w	c,s1
mul.s	s1,s0	ld.w	b,s2
ld.w	b,s1	mul.s	s0,s1
add.s	s1,s0	ld.w	e,s0
ld.w	e,s1	ld.w	f,s3
div.s	s1,s0	add.s	s1,s2
ld.w	f,s1	div.s	s0,s2
sub.s	s1,s0	sub.s	s3,s2
st.w	s0,a	st.w	s2,a

In the original code, operations execute one at a time. In the optimized code, groups of registers used by different functional units can be loaded simultaneously. All registers are loaded before arithmetic begins, if possible. Operations such as add and divide that employ different functional units also execute simultaneously. The concurrent execution of machine instructions on multiple functional units is distinct from parallel processing, which takes place on multiple processors.

For more information on functional units, see *CONVEX Architecture Reference.*

Register Allocation

CONVEX C uses an allocation technique that fully exploits the CONVEX register set. By maximizing the number of registers used for each expression, the compiler allows more register loads to be grouped together for concurrent execution ("pipelining") and reduces the number of register conflicts.

Span-Dependent Instructions

Whenever possible, the compiler generates short-form instructions for conditional and unconditional jumps and branches. These 2-byte branch and 4-byte jump instructions conserve memory and increase execution speed. They are generated when the span, or distance from the branch or jump instruction to its target location, is within defined limits for these instructions.

For more information on jump and branch instructions, see *CONVEX Architecture Reference* and *CONVEX Assembly-Language User's Guide.*

Tree-Height Reduction

Expressions are represented internally as trees, the height of which corresponds to the depth of the expression. These trees are optimized by a transformation called *balancing* or *tree-height reduction.* Consider, for example, an integer expression

```
a + b + c + d + e + f + g + h
```

This expression can be evaluated as

```
Order 1:  (a+ (b+ (c+ (d+ (e+ (f+ (g+h)))))))   /* Items inside () are evaluated first */
```

or

```
Order 2:  (((a+b) + (c+d)) + ((e+f) + (g+h)))
```

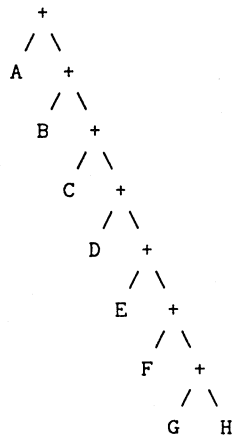
Order 1 evaluates `(g + h)` first. No two additions can be carried out simultaneously, because each addition depends on the result of the addition to its right.

Order 2 allows four additions to execute simultaneously by overlapping instruction execution on multiple functional units. `(a + b)`, `(c + d)`, `(e + f)`, and `(g + h)` can be computed simultaneously. None of these additions requires a result from any other addition. When these four additions are finished, the simultaneous calculation of `((a + b) + (c + d))` and `((e + f) + (g + h))` is also possible.

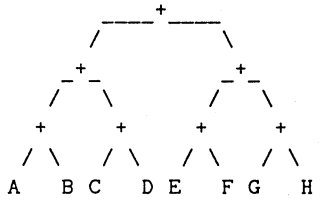
Items inside parentheses are evaluated first.

The internal representations of these evaluation orders are

Order 1



Order 2



In general, the time required to evaluate an expression is proportional to the depth of the tree, or the deepest nesting level of parentheses. The depth of the tree is seven if Order 1 is used, but only three if Order 2 is used. Whenever possible, the compiler chooses an evaluation order to minimize the depth of an expression and maximize the use of functional units. This optimization frees you from the burden of trying to write expressions in an order that ensures the most efficient execution.

Note

This transformation is performed on floating-point expressions only if the program is compiled with the `-parens explicit` or `-parens ignore` option. For programs compiled in `-pcc` or `-ext` mode, `-parens ignore` is the default. For programs compiled in other modes, `-parens explicit` or `-parens ignore` must be requested for reordering to occur.

The `-parens implicit` option prevents the compiler from reordering floating-point expressions. When `-parens implicit` is used, the compiler evaluates floating-point expressions according to a strict interpretation of grammar and associativity rules. This option has no effect on integer expressions, which are always subject to reordering.

Optimizations Performed at -O0

At optimization level -O0, the compiler performs machine-independent scalar optimizations within the scope of a basic block (a linear sequence of statements with a single entry and a single exit). In addition, the compiler continues to perform the machine-dependent optimizations introduced at -no.

The following subsections describe the new optimizations that the compiler performs at -O0.

Algebraic and Trigonometric Simplification

The compiler performs the algebraic and trigonometric simplifications shown in the following table.

Original expression	Simplified expression
x + 0	x
x * 1	x
x * 0	0
x - 0	x
x && 1	x ? 1 : 0
x && 0	0
x || 1	1
x || 0	x ? 1 : 0
x & 0	0
x | 0	x
-1 * x	-x
x - x	0
x / -1	-x
x / x	1
0 - x	-x
0 / x	0
sin(x) * cos(x)	0.5 * sin(2*x)
sin(x) / cos(x)	tan(x)
cos(x) / sin(x)	1 / tan(x)

Obvious variants of these operations are performed for the commutative operators. For example, x + 0 + y is converted to x + y. Trigonometric simplifications are performed only if the fastmath.h header file is specified. They are not performed if the math.h header file is used instead.

Assignment Substitution

Assignment substitution eliminates redundant loads. The compiler retains the value assigned to a variable and replaces subsequent references to that variable with the assigned value. Consider this example:

```
x = y + c;
  = ... x ...;
  = ... x ...;
x = 4.179;
```

After the first statement is executed, the value `y + c` remains in a register. Subsequent references to `x` are replaced by references to the register until the value of `x` is changed or the end of the basic block is reached. This eliminates repeated loads of `x` into a register, which saves time and increases opportunities for further optimization. In the example above, assignment substitution makes the first assignment to `x` redundant. The compiler eliminates it:

```
reg = y + c;
  = ... reg ...;
  = ... reg ...;
x = 4.179;
```

Because of assignment substitution it is rarely necessary or possible to hand-optimize a program by replacing a variable reference with a constant.

Constant Propagation and Folding

Constant propagation is a form of assignment substitution. After assigning a constant to a variable, the compiler replaces subsequent references to that variable with a constant. For example, if you assign `x = 5`, `x` is replaced with the constant 5 wherever it occurs within that basic block or until a new value is assigned to `x`.

The compiler can also replace operations performed on constants, such as `y = 5 + 7`, with the computed value (in this case, 12). This is called constant folding. The constant value (12) can then be propagated to replace future references to `y` within the basic block.

Original Code	Optimized Code

`i= 5;`
`j= 0;`
`...`
`j= j + 2;`
`...`
`k= k + i*j;`
	`i= 5;`
	`/* assignment eliminated */`
	`...`
	`j= 2;`
	`...`
	`k= k + 10;`

If the fastmath.h file is used, the most frequently used math functions are folded when applied to constant arguments. For example, `sin(1.0)` becomes `1`. If the math.h header file is used instead, this optimization is not performed.

If an integer overflow occurs as the result of constant folding, the compiler reports "Integer constant truncation." If a floating-point overflow occurs, the compiler reports "Real constant either too large or too small." Floating-point underflow always results in zero. If any of these messages occur, modify your source code to eliminate the offending operation or bring the value of the constant within acceptable bounds.

Elimination of Common Subexpressions

The compiler can recognize a subexpression that is repeated within a basic block. The value of the common subexpression is retained in a register to eliminate redundant calculations and register loads. For example, the compiler recognizes `b + c` as a common subexpression of `a + b + c + d` and `b + c + e` and calculates the value only once.

Calculation of array addresses may also result in common subexpressions that the compiler eliminates.

Instruction Scheduling

At optimization level `-O0` and above, instructions from multiple statements may be rescheduled. Compare two assembly codes for the following statements:

```
t= b + c*d;
a= t/e - f;
```

Original Code		Optimized Code	
ld.w	d,s0	ld.w	d,s0
ld.w	c,s1	ld.w	c,s1
mul.s	s1,s0	ld.w	b,s2
ld.w	b,s1	mul.s	s0,s1
add.s	s1,s0	ld.w	e,s0
st.w	s0,t	ld.w	f,s3
ld.w	e,s1	add.s	s1,s2
div.s	s1,s0	st.w	s2,t
ld.w	f,s1	div.s	s0,s2
sub.s	s1,s0	sub.s	s3,s2
st.w	s0,a	st.w	s2,a

In the code on the left, which is compiled at `-no`, instructions from each statement are scheduled independently. All instructions derived from the first statement execute first. Instructions derived from the second statement execute after that. In the optimized code on the right, instructions from the two statements are scheduled together, as if they were derived from a single statement.

Redundant-Assignment Elimination

This optimization removes unnecessary assignments to a variable. When a variable is not used between two assignments, the first assignment is eliminated. The following code contains a redundant assignment to x:

```
x = y + c;
    ...        /* x not used */
x = 3.1416;
    ...
y = (x+7) * ...;
```

The compiler removes the assignment and the code is optimized to:

```
    ...        /* x not used */
x = 3.1416;
    ...
y = (x+7) * ...;
```

Redundant-Use Elimination

Redundant-use elimination is a special case of common subexpression elimination. Here, the common subexpression is a variable. The compiler detects multiple references to a variable that occur between assignments and eliminates unnecessary register loads by retaining the value in a register.

Strength Reduction

In some instances, the compiler can replace an arithmetic operation with an equivalent operation that has lower strength and lower cost. For example, the compiler transforms integer multiplication by 2, 4, 8, or 16 into integer shifts:

a * 2 becomes a<< 1.
a * 3 becomes a + (a<<1).

Integer divisions cannot be strength-reduced because shifting a negative number produces an incorrect result:

a / 2 remains a / 2.

Multiplication involving real numbers can be reduced to addition. Division by a constant is reduced to multiplication:

x * 2 becomes x + x.
x / C becomes x * K where K= 1 / C.

Because C is a constant, K is also a constant, which can be precomputed at compile time.

Optimizations Performed at -O1

Global optimization is performed across a group of basic blocks but within the scope of a C function. The -O1 option on the cc command line causes the compiler to perform global, as well as basic-block and machine-dependent, optimization.

The following subsections describe global optimizations the compiler performs.

Constant Propagation and Folding

Constant propagation and folding at the global level is similar to constant propagation and folding within a basic block. However, the scope of the optimization is now an entire C function.

Original Code	Optimized Code

`main()`
`{`
`int a, b, c, i;`
`a= 5;`
`b= 15;`
`scanf("%d", &i);`
`if (i<0)`
`{`
`a = 6;`
`c = a;`
`}`
`else`
`{`
`c = a + b;`
`b = a + 8 + c;`
`}`
`printf("%d %d %d", a, b, c);`
`}`
	`main()`
	`{`
	`int a, b, c, i;`
	`a= 5;`
	`b= 15;`
	`scanf("%d", &i);`
	`if (i<0)`
	`{`
	`a = 6;`
	`c = 6;  /* a = 6 */`
	`}`
	`else`
	`{`
	`c = 20;  /* a = 5, b = 15 */`
	`b = 33;  /* a = 5, c = 20 */`
	`}`
	`printf("%d %d %d", a, b, c);`
	`}`

Programmers who are unfamiliar with optimizing compilers frequently try to propagate constants by hand in order to generate more efficient code. The CONVEX C compiler completely eliminates the need for hand-propagation for programs compiled at -O1 or higher.

Code Motion

Code motion is the movement of invariant computations performed inside a loop out of the loop. An invariant computation yields the same result on every iteration of a loop. If an invariant expression does not lie on a path to all loop exits, it is not moved out of the loop unless you use the -uo (unsafe optimizations) option.

Original Code	Optimized Code

`main()`
`{`
`float ar[10];`
`float a, b, c, e;`
`int i;`
`...`
`for (i= 0; i<10; i++)`
`{`
`a= b+ (c*4)/ (-(e*b)+ sqrt(c));`
`ar[i] = a + b * c;`
`}`
`...`
`}`
	`main()`
	`{`
	`float ar[10];`
	`float a, b, c, e;`
	`int i;`
	`...`
	`a = b + (c * 4) / (-(e * b) + sqrt(c));`
	`t1 = a + b * c;`
	`for (i= 0; i<10; i++)`
	`ar[i] = t1;`
	`...`
	`}`

All variables used in the assignments above are invariant—they do not change their values—within the scope of the loop. The compiler recognizes this and moves the calculations and assignment to the variable `a` out of the loop. These relatively costly calculations are now performed only once, and, at higher optimization levels, the loop can be vectorized.

Copy Propagation

The compiler can sometimes replace a variable with another variable to which it has been equated. This is called copy propagation. For example, the statement `x = y` may allow the compiler to replace later occurrences of `x` with `y`.

Dead Code Elimination

As a result of constant propagation and folding, the arithmetic or logical expression of an `if` statement may be folded to `true` or `false`. The unreachable alternative, called "dead code," is eliminated with the `if` test.

Elimination of Common Subexpressions

Subexpression elimination also occurs at the global (function) level. The value of the common subexpression is retained in a register if one is available; otherwise, it is assigned to a temporary variable. Subsequent occurrences of the common subexpression are replaced by references to the register or temporary variable.

Original Code	Optimized Code

`main()`
`{`
`...`
`a= b + (c*4)/ (-(j*b)+ sqrt(c));`
`if (k<1)`
`m = 5;`
`else`
`m = 6;`
`f= e - (b + (c*4)/(-(j*b) + sqrt(c)));`
`}`
	`main()`
	`{`
	`...`
	`t1= b + (c*4)/ (-(j*b)+ sqrt(c));`
	`a = t1;`
	`if (k<1)`
	`m = 5;`
	`else`
	`m = 6;`
	`f= e - t1;`
	`}`

The compiler recognizes that the subexpression is used both before and after the `if` statement. It saves the value of the subexpression in the temporary variable `t1` before the `if` statement and uses this variable later to compute the value of `f`.

Hoisting Scalar and Array References

The compiler can sometimes "hoist," or lift, a scalar or array reference out of a loop. By moving the reference out of the loop, hoisting eliminates redundant operations. At optimization level `-O1`, hoisting occurs when the value of a scalar variable or array element is unchanged within the loop. At optimization level `-O2`, hoisting of an array reference occurs if the array is indexed only by loop constants and the loop control variable.

In the following example, substitution is performed if the compiler determines that `x` and `y` are unchanged between the assignment and the reference. This means that the assignment to `x` is redundant and can be eliminated.

```
x = y;
...
w = z - x;
```

becomes

```
...
w = z - y;
```

Redundant-Assignment Elimination

Redundant assignment elimination removes assignments to variables that are not referenced elsewhere within a function.

Original Code	Optimized Code

`main()`
`{`
`  ...`
`x= y*z;`
`if (a>0)`
`  {`
`  ...`
`  /* x not referenced */`
`  x= a*y + ...`
`  }`
`else`
`  {`
`  ...`
`  /* x not referenced */`
`  x = y*z+ ...`
`  }`
`...`
`/* a, x not referenced */`
`}`
	`main()`
	`{`
	`...`
	`if (a>0)`
	`  {`
	`  ...`
	`  }`
	`else`
	`  {`
	`  ...`
	`  }`
	`}`

Assignments to objects referenced by pointers and assignments to global variables are never eliminated. If an assignment statement contains a function call, the assignment is eliminated, but the function call is retained:

Original Code	Optimized Code

`main()`
`{`
`  ...`
`  i= intfun();`
`  ...`
`  /* i not used */`
`  }`
	`main()`
	`{`
	`  ...`
	`  (void) intfun();`
	`  ...`
	`}`

If a function has no side effects, the function call can be eliminated also, saving much more time. Function calls are eliminated only if you explicitly requests their elimination with the `no_side_effects` directive. The form of this directive is

```
/*$dir no_side_effects (func_name_list)*/
```

where `func_name_list` is a list of one or more function names, separated by commas. The `no_side_effects` directive must precede the occurrence of the function call to be eliminated.

CAUTION The `no_side_effects` directive should not be used with the name of any function that:

- Changes the value of a global variable.
- Does input or output.
- Calls another function that is not specified as `no_side_effects`.

Strength Reduction and Loops

Strength reduction can optimize operations on loop-induction variables and loop constants. Frequent candidates for strength reduction include multiplications such as those used to calculate the address of a subscripted variable within a loop.

Strength reduction is not performed on operations that involve one or more `float` operands. Because floating-point arithmetic is imprecise for large numbers, reduced operations may not yield the same results as nonreduced operations. If an expression does not lie on a path to all loop exits, it is not reduced unless you use the `-uo` option.

Original Code

```
main()
{
 i = 1;          /* i is an induction */
 while (i<=100)          /* variable */
   {
    x = i * c;  /* c is loop invariant */
    ...
    i+= 2;
   }
    ...
}
```

Optimized Code

```
main()
{
 i = 1;
 t1= c;
 t2 = 2 * c;
 while (i<=100)
   {
    x=t1;
    ...
    t1+= t2;
    i+= 2;
   }
 ...
}
```

The compiler recognizes that `i` is incremented by 2 on each iteration. It infers that `x` is incremented by `2 * c`, a loop constant. Instead of calculating `i * c` on every iteration, the compiler produces code that calculates `2 * c` only once and increments `x` by the value saved in `t2`.

Conclusion

The CONVEX C compiler automatically optimizes scalar code for speed and efficiency. When it is unable to determine the safety of an optimization, the compiler does not transform code. Potentially unsafe optimizations can be done if you use the `-uo` option.

Chapter 3
Vector Optimization

Vectorization converts scalar operations on array elements into equivalent vector operations that use vector registers to perform simultaneous computations on up to 128 elements of an array. To produce efficient vector code, the compiler performs several additional optimizations. For example, the compiler partitions arrays with more than 128 elements into strips with no more than 128 elements each, and vectorizes operations on each strip. This optimization is called strip-mining.

The `-O2` option on the `cc` command line tells the compiler to generate vector code. It also tells the compiler to perform whatever transformations are necessary to generate efficient vector code, in addition to the scalar optimizations performed at `-O1`.

Basic Operation

An inner loop typically performs identical operations on multiple elements of an array or arrays. Each instruction in the loop is executed on every iteration. Consider the following loop:

```
for (i=0; i<100; i++)
  a[i] = b[i] + c[i];
```

Without optimization, this loop requires at least 700 scalar instruction executions. (Load an element of `B` and an element of `C`, add them, store the result in the corresponding element of `A`, then load, increment, and store `I`. Repeat for the next 99 elements.)

At `-O2`, the compiler generates vector code that loads 100 elements of `b` into a vector register, loads 100 elements of `c` into another vector register, adds the two registers, and stores the resulting elements in `a`. Think of the resulting vector code as a single statement with only four instructions (vector load, vector load, add, and store):

```
a[0:99] = b[0:99] + c[0:99];    /* Pseudocode for vector add */
```

Transformations the Compiler Performs

At `-O2`, the compiler rearranges statements and instructions to increase the number of loops that can be vectorized. The following subsections explain the most important of these vectorizing transformations.

> **Note**
>
> The standard C rules for evaluating expressions, used by default in the `-std` and `-str` compilation modes and invoked by the `-paren implicit` option in other modes, interfere with the vectorization of many loops. Using `-parens explicit` or `-parens ignore` may increase vectorization, at the cost of some accuracy.

Strip-Mining

CONVEX vector registers can hold up to 128 elements. When the iteration count (trip count) of a vectorizable loop exceeds 128, the compiler strip-mines the loop before vectorizing it. Strip-mining converts the original loop into two nested loops. The inner loop has an iteration count less than or equal to 128; the outer loop serves as a driver, which determines the number of times the inner loop is executed.

Original Loop

```
for (i = 0; i<n; i++)
  a[i] = b[i] + c[i];
```

Vectorized Loop

```
i = 0;
for (i_outer = n; i_outer>0; i-=128)
  {
   for (i_inner = i; i_inner<=i + min(127, i_outer-1), i++)
     a[i_inner] = b[i_inner] + c[i_inner];
   i += 128;
  }
```

In the vectorized code, `i_outer` is a variable created by the compiler to count the number of elements remaining to be processed. The `i_inner` loop represents a vector operation. `i` holds the starting index for each vector operation.

If n = 300, i_outer is tested 4 times. The following table shows values of i and i_outer each time i_outer is tested, and the elements of a, b, and c that are processed.

i	i_outer	Elements Processed
1	300	0...127
129	172	128...255
257	44	256...299
385	-84	—

The fourth test of i_outer fails (i_outer < 0). The loop terminates and no additional elements of the arrays are processed.

Loop Distribution

Inner loops can be vectorized and strip-mined (if necessary) without any additional transformations; outer loops cannot. However, the compiler can often vectorize an outer loop by *distributing* the outer loop into a number of smaller loops called *distributed parts.* It can then vectorize one or more of the distributed parts.

Original Loop

```
for (i=0; i<n; i++)
  {
   b[0][i] = 0;
   for (j = 0; j<m; j++)
      a[i] = a[i] + b[j][i] * c[j][i];
   d[i] = e[i] + a[i];
  }
```

Vectorized Loop

```
for (i=0; i<n; i++)
  b[0][i] = 0;

for (i = 0; i<n; i++)
  for (j = 0; j<m; j++)
    a[i] = a[i] + b[j][i] * c[j][i];

for (i=0; i<n; i++)
    d[i]= e[i] +a[i];
```

In this example, three distributed parts of the i loop are created. The nested j loop is separated from the assignments to arrays b and d, which are now performed by inner loops. As a result, the compiler can vectorize the operations that assign to b and d, as well as the original inner loop that assigns to a.

Loop Interchange

The compiler can interchange loops to improve performance. The following matrix addition shows this transformation.

Original Loop	Vectorized Loop

`for (i=0; i<n; i++)`
`  for (j= 0; j<m; j++)`
`  a[i][j] = b[i][j] + c[i][j];`
	`for (j =0; j<m; j++)`
	`  for (i=0; i<n; i++)`
	`  a[i][j] = b[i][j] + c[i][j];`

The compiler interchanges the `i` and `j` loops so that elements of `b` and `c` that are contiguous in memory are loaded into vector registers. This optimization significantly improves performance over the column-wise approach in the original source.

Achieving more efficient array accesses is not the only reason the compiler may interchange nested loops. Loops may also be interchanged to achieve a more efficient vector length on the inner loop. This is illustrated in the following example:

Original Loop	Vectorized Loop

`for (i=0; i<4000; i++)`
`  for (j= 0; j<15; j++)`
`  a[j][i] = b[j][i] + c[j][i];`
	`for (j =0; j<15; j++)`
	`  for (i=0; i<4000; i++)`
	`  a[j][i] = b[j][i] + c[j][i];`

Here, the subscript controlled by the inner loop (`j`) varies over a small range (0 to 15). It is possible for the compiler to vectorize this loop, but doing so would yield a short vector length of 15. The compiler weighs the benefit of efficient memory access against the benefit of a longer vector length. In this example, it sacrifices efficient memory accesses and interchanges the two loops to achieve a longer vector length. (If loops like this occur often in your program, you should consider rewriting the code to reverse the dimensions of the array.)

Paired Hoist and Sink

A vector register can sometimes be used as an accumulator, making it possible for the compiler to hoist and sink loads and stores of the register outside the vector loop. Hoisting moves an operation, such as a register load, from a loop to a basic block preceding the loop. Sinking is the complement of hoisting: it moves an operation, such as a register store, from a loop to a basic block following the loop. Consider the following example:

```
for (j=0; j<n; j++)
  for (i=0; i<n; i++)
    a[i] = a[i] * b[i][j];
```

When this program fragment is compiled at `-O2`, the `i` loop is vectorized. In addition, the load of vector `a` is hoisted above the loop, while the store of vector `a` is sunk below it. This eliminates the need for repeated vector loads and stores and makes the vector loop even faster.

Vector loads and stores can be hoisted and sunk only if:

- The array reference and array assignment have the same subscript.
- All subscripts of the array are vector subscripts or loop constants.

The compiler may interchange loops to make a nonvector subscript a loop constant.

Conditional Induction Variables

An *induction variable* is one whose value is conditionally or unconditionally incremented or decremented with each iteration of a loop. Loop-control variables are usually induction variables, but not all induction variables are loop-control variables. For example, in the following loop, `k` is an induction variable but not a control variable in the following loop:

```
k = 0;
for (i = 0; i<100; i++)
  if (cond(i))
    {
     a[i] = b[k];
     c[k] = d[i];
     k++;
    }
```

Because the value of `k` is incremented on each particular iteration only if the outcome of some conditional test is `true`, `k` is called a *conditional induction variable.* The compiler can often recognize conditional induction variables and generate vector code for the expressions in which they occur.

In the preceding example, the compiler generates code that determines values of `i` for which `cond(i)` is `true` (not zero) and stores the truth values in an array. The first K elements (where K represents the value of `k` on completion of the loop) of array `b` are loaded into a vector register. Using the truth values contained in the `cond` array as a guide, the vector is expanded to 100 elements. This expanded vector is then stored, using the `cond` array as a mask, into the first 100 elements of array `a`. The first 100 elements of `d[1:100]` are loaded into a vector register. The vector is compressed to length K, using the `cond` array as a guide, and then stored into the first K elements of `c`.

A single machine instruction determines the truth values of `cond(i)`; expansion and compression are single instructions as well, so this optimization markedly improves code performance. (For information on the `merg` and `cprs.t` instructions, which perform the operations just described, see *CONVEX Assembly-Language User's Guide* and *CONVEX Architecture Reference.*)

Inhibitors of Vectorization

Any of the following conditions can inhibit or prevent vectorization of a loop:

- Multiple entries into or exits from the loop.
- A conditional test (`if` statement or ?: operator) in the loop.
- A function call in the loop.
- A recurrence.

The first three items in this list are fairly easy to avoid and are discussed in Chapter 6, "Efficient Programming Constructs." Recurrence, however, is one of the most frequent inhibitors of vectorization. The following section talks about recurrence and its effects on vectorization.

Recurrence

A value calculated in one iteration of a loop may be referenced in another iteration. When this happens, the value *recurs* and a *recurrence* exists. (A recurrence is sometimes called a recursion. To avoid confusion, this guide uses only the term recurrence when referring to loops. The term recursion refers only to function recursion.)

Recurrence is closely related to *data dependency.* A data dependency is a relationship between two operations such that one operation depends on the results of the other. This implies a definite time order of operations: execution of one operation must always precede execution of the other, and the execution order cannot be changed without affecting results.

Dependencies may be *loop-carried* or *loop-independent.* Only loop-carried dependencies produce recurrences.

Some loops are written in such a way that the compiler cannot determine whether a recurrence exists. A possible recurrence that does not actually exist is called an *apparent* recurrence. The compiler does not automatically vectorize a loop that contains a real or apparent recurrence.

A loop-carried dependency (LCD) exists when one iteration of a loop computes a value that is referenced on another iteration. Consider the following example:

```
for (i=0; i<n; i++)
  arra[i+1] = arra[i] + 3.14;
```

The dependency is *carried* by the loop from one iteration to the next. If the statement `arra[i+1] = arra[i] + 3.14` occurs outside a loop, no dependency exists.

Dependencies may be *backward* or *forward*. A backward LCD exists when one iteration references a variable whose value is assigned on a previous iteration. The example above contains a backward LCD. The first iteration of the loop assigns a value to `arra[2]`; the second iteration references this value and assigns a new value to `arra[3]`, and so on. The iterations of the loop are inherently serial; the loop cannot be vectorized.

A forward LCD exists when one iteration references a variable whose value is assigned on a later iteration. This is also called an *antidependency*. The following loop contains a forward dependency:

```
for (i=0; i<n; i++)
  arra[i] = arra[i+1] + 3.14;
```

In this example, the first iteration assigns a value to `arra[1]` and references `arra[2]`; the second iteration assigns a value to `arra[2]` and references `arra[3]`. The reference to `arra[i]` depends on the fact that the `i+1`th iteration, which assigns a new value to `arra[i]`, has not executed previously. (The `i+1`th iteration must take place simultaneously with or later than the `i`th iteration.) A forward dependency, therefore, does not prevent vectorization of a loop.

The compiler can vectorize some loops that have backward LCDs. Consider the following example:

```
for (i=0; i<n; i++)
  {
   arra[i] = arrb[i] + arrc[i];
   arrb[i+1] = arrd[i] * 3.14;
  }
```

In this loop, the assignment to `arra[2]` on the second iteration depends on the value assigned to `arrb[2]` on the first iteration. The compiler interchanges the statements within the loop so that the assignment to `arrb` takes place before the assignment to `arra`:

```
for (i=0; i<n; i++)
  {
   arrb[i+1] = arrd[i] * 3.14;
   arra[i] = arrb[i] + arrc[i];
  }
```

The compiler can now vectorize both statements contained in this loop.

When an LCD is caused by a scalar variable, the compiler can eliminate the recurrence through a transformation called *scalar spreading*. Within the body of a loop, the compiler replaces all occurrences of a scalar variable causing a recurrence with a temporary vector. The correct value is assigned to the scalar when the loop terminates.

Original Loop	Vectorized Loop

`for (i=0; i<10; i++)`
`  {`
`  x = arra[i];`
`  arrg[i]+= b* x;`
`  }`
	`for (i=0; i<10; i++)`
	`  {`
	`  tva[i] = arra[i];`
	`  arrg[i]+= b* tva[i];`
	`  }`
	`x = arra[9];`

In this example, the temporary vector `tva` replaces all occurrences of `x` in the body of the loop. When the loop terminates, the value of `arra[9]` is assigned to `x`.

A backward LCD that cannot be eliminated may not stop vectorization completely. Using temporary vectors, the compiler can sometimes vectorize part of a loop that contains an LCD. The following loop cannot be fully vectorized because of a backward LCD:

Original Loop	Vectorized Loop

`for (i=1; i<n; i++)`
`  arra[i] = arra[i-1] + arrb[i] * arrc[i];`
	`for (i=1; i<n; i++)`
	`  tva[i] = arrb[i] * arrc[i];`
	
	`for (i = 1; i< n; i++)`
	`  arra[i] = arra[i-1] + tva[i];`

The assignment to `arra[i]` depends on the value of `arra[i-1]`, which is computed on the previous iteration. The compiler isolates the dependency by distributing the loop and vectorizes the first distributed part. This transformation is called *partial vectorization* because it distributes a loop into vector and scalar parts.

A loop-independent dependency (LID) exists when two statements must be executed in a specific order within a single iteration. The dependency between statements would exist even if the statements occurred outside the loop. The following loop contains two such LIDs:

```
for (i = 0; i<n; i++)
  {
   arra[i] = arrb[i] + arrc[i];  /* statement 1 */
   arrb[i] = 0;                  /* statement 2 */
   arrc[i]++;                    /* statement 3 */
  }
```

Here, statement 1 can be properly evaluated only if statements 2 and 3 have not yet been evaluated. Statement 1 is antidependent on statements 2 and 3. A forward LID exists between statements 1 and 2; another exists between statements 2 and 3.

A backward LID cannot exist except as the result of a logic error. It occurs when the first of two sequential statements can produce the proper result only if it follows the second.

Note

Loop-independent dependencies do not normally prevent vectorization of a loop. Loop-carried dependencies, which cause recurrences, can prevent vectorization. Vectorization is inhibited when an LCD between an assignment and a reference to an array prevents the compiler from generating correct vector code.

An LID can stop vectorization, however, by preventing the compiler from eliminating an LCD. The following loop, for example, cannot be vectorized:

```
for (i = 1; i<n; i--)
  {
   arra[i] = arrb[i] - arrc[i];    /* statement 1 */
   arrb[i+1] = arra[i] + arrd[i];  /* statement 2 */
  }
```

Interchanging the two statements within the loop would remove the LCD that exists between the assignment to `arrb[i+1]` in statement 2 and the reference to `arrb[i]` in statement 1. The LID between the assignment to `arra[i]` in statement 1 and the reference to `arra[i]` in statement 2 prevents this interchange.

An *apparent recurrence* exists when the compiler lacks sufficient information to prove that an actual recurrence does not exist. Apparent recurrences usually result from using an array subscript that is:

- An array reference.
- A loop invariant of unknown sign.
- Use of pointers (see Chapter 7 on *aliasing*).

The following loop cannot be vectorized because the compiler does not know the values that `arrj[i]` might take, and thus cannot determine whether a dependency exists between the assignment and reference to `arra[arrj[i]]`. To avoid possible wrong answers, the compiler assumes that a recurrence exists.

```
for (i=1; i<n; i++)
  arra[arrj[i]] = arra[arrj[i+1]];
```

You can correct many apparent recurrences using the `no_recurrence` directive or by modifying your code. The following loop is similar to the one shown in the section on paired vector hoist and sink. In this case, however, the loop cannot be vectorized because the sign of `k` is unknown:

```
if (k>=0)
  for (j = 0; j < n; j++)
    for (i = 0; i < n; i++)
      arra[i] = arra[i+k] * arrb[i][j];
```

From the context, you know that the value of `k` is always positive (or zero) within this loop. Unless a constant value has been assigned to `k` and propagated, the compiler does not know this. Because the compiler does not make any assumptions about the value of `k`, it appears that the value of `arra[i+k]` could depend on an assignment to `arra[i]` in a previous iteration. For this reason, the compiler assumes a dependency that does not exist.

With the `no_recurrence` directive, the compiler can vectorize this loop:

```
if (k>=0)
  for (j = 0; j < n; j++)
    /*$dir no_recurrence*/
    for (i = 0; i < n; i++)
      arra[i] = arra[i+k] * arrb[i][j];
```

Apparent recurrences happen only with array references. With scalar variables, the compiler can always determine whether a recurrence exists or not. The compiler's optimization report can tell you what variable is causing a recurrence.

CAUTION A `no_recurrence` directive on a loop containing an actual recurrence can cause your program to produce incorrect results. Check your program by hand to make sure no actual recurrence exists before using a `no_recurrence` directive. Always check the output from the program after using the `no_recurrence` directive to make sure it still yields correct results. (Even if there is no actual recurrence, the output may change slightly due to vectorization.)

Reductions

Instructions built into the CONVEX vector hardware allow the compiler to vectorize a special class of recurrence known as *reductions*. In general, a reduction has the form

```
x = x operator y
```

where

`x`	is a variable that is not assigned or referenced elsewhere in the loop
`y`	is any loop-constant (loop-invariant) expression not involving `x`
`operator`	is one of +, −, *, &, |, ==, and !=

The compiler cannot vectorize a reduction if a type conversion occurs within the loop. This happens in the following example:

```
float sum, arra[1000];
int i;
sum = 0.0;

for (i=0; i<n; i++)
  sum += arra[i]* 2.0;
```

Multiplying `arra[i]` by `2.0` (a floating-point constant of type `double`) causes a type conversion, which prevents vectorization.

The Optimization Report

For functions compiled at optimization levels `-O2` and `-O3`, the compiler produces an *optimization report* describing the optimizations performed. The `-or` option determines what tables the report comprises:

Chapter 4
Parallel Optimization

At optimization level -O3, the CONVEX C compiler vectorizes and parallelizes programs to enhance performance. Unlike vectorization, parallelization does not reduce CPU time. Instead, it spreads processing of a single program across multiple CPUs, improving the program's turnaround time (also known as *wall-clock time*).

Basic Operation

Parallelization divides a program into *threads*. A thread is a sequence of instructions that must execute on a single CPU. Parallelism can be classified according to the size and number of these threads. A program with a few, large threads is said to be *coarse-grained*. A program with many, small threads is said to be *fine-grained*. This property is called *granularity*. Depending on the method of parallelization, the size of a thread can range from half a program (coarse) to a single source statement (fine-grained). The CONVEX C compiler parallelizes a program at the loop level, provided a medium-grained parallelism. The compiler vectorizes inner loops, but parallelizes outer loops. Often, these outer loops are the strip-mine loops created when an inner loop is vectorized. Just as for vectorization, the compiler distributes and interchanges loops to produce the most efficient parallel code. The compiler can also parallelize most scalar reductions and assignments by adding synchronization code.

Consider the following matrix multiplication, which is compiled at -O3:

```
for (i=0; i<n; ++i)
  for (j=0; j<n; ++j)
    {
     arrc[i][j]= 0.0;
     for (k=0; k<n; ++k)
       arrc[i][j]= arrc[i][j] + arra[i][k] * arrb[k][j];
    }
```

The compiler distributes the i and j loops:

```
for (i=0; i<n; ++i)
  for (j=0; j<n; ++j)
    arrc[i][j]= 0.0;

for (i=0; i<n; ++i)
  for (j=0; j<n; ++j)
     for (k=0; k<n; ++k)
       arrc[i][j]= arrc[i][j] + arra[i][k] * arrb[k][j];
```

To maximize efficiency of array accesses, the compiler interchanges the j and k loops in the second nest:

```
for (i=0; i<n; ++i)
  for (j=0; j<n; ++j)
    arrc[i][j]= 0.0;

for (i=0; i<n; ++i)
  for (k=0; k<n; ++k)
     for (j=0; j<n; ++j)
       arrc[i][j]= arrc[i][j] + arra[i][k] * arrb[k][j];
```

Both j loops are strip-mined to the optimum vector length, which is a function of n:

```
for (i=0; i<n; ++i)
  {
   vlen= optimum(n);                                  /* set vector length */
   for (j_outer=1; j_outer<n; j_outer+= vlen)         /* strip-mine */
     for (j= j_outer; j< min(n, j_outer+ vlen-1)
       arrc[i][j]= 0.0;
  }

for (i=0; i<n; ++i)
  for (k=0; k<n; ++k)
    for (j_outer=0; j_outer<n; j_outer+= vlen-1)     /* strip-mine */
      for (j= j_outer; j< min(n, j_outer+ vlen-1)
        arrc[i][j]= arrc[i][j] + arra[i][k] * arrb[k][j];
```

The compiler recognizes the induction variable j_outer as invariant within the k loop of the second nest and compiler interchanges the j_outer and k loops:

```
for (i=0; i<n; ++i)
  {
   vlen= optimum(n);
   for (j_outer=1; j_outer<n; j_outer+= vlen)         /* strip-mine */
     for (j= j_outer; j< min(n, j_outer+ vlen-1); ++j)
       arrc[i][j]= 0.0;
  }

for (i=0; i<n; ++i)
  for (j_outer=0; j_outer<n; j_outer+= vlen-1)       /* strip-mine */
    for (k=0; k<n; ++k)
      for (j= j_outer; j< min(n, j_outer+ vlen-1); ++j)
        arrc[i][j]= arrc[i][j] + arra[i][k] * arrb[k][j];
```

The compiler parallelizes outer loops and vectorizes inner loops. (The notation [first:last] for array subscripts means all elements of the array with subscripts from first to last. V0 and V1 represent vector registers.)

```
for (i=0; i<n; ++i)                                        /* PARALLEL */
  {
   vlen= optimum(n);
   for (j_outer=1; j_outer<n; j_outer+= vlen)              /* strip-mine */
     {
       V0= 0.0;                                          /* vector clear */
       arrc[j_outer:min(n, j_outer+ vlen-1)][i]= V0;     /* vector store */
     }
  }

for (i=0; i<n; ++i)                                        /* PARALLEL */
  for (j_outer=0; j_outer<n; j_outer+= vlen-1)             /* strip-mine */
    for (k=0; k<n; ++k)
      {
       V0= arrc[j_outer:min(n, j_outer+ vlen-1)][i];     /* vector load */
       V1= arra[j_outer:min(n, j_outer+ vlen-1)][k];     /* vector load */
       V1= V1* arrb[k][j];                               /* vector multiply */
       V0= V0+ V1;                                       /* vector add */
       arrc[j_outer:min(n, j_outer+ vlen-1)][i]= V0;     /* vector store */
      }
```

The compiler hoists and sinks a vector load and store out of the k loop. This allows V0 to be used as an accumulator register, resulting in much faster vector operations.

```
for (i=0; i<n; ++i)                                        /* PARALLEL */
  {
   vlen= optimum(n);
   for (j_outer=1; j_outer<n; j_outer+= vlen)
     {
       V0= 0.0;
       arrc[j_outer:min(n, j_outer+ vlen-1)][i]= V0;
     }
  }

for (i=0; i<n; ++i)                                        /* PARALLEL */
  for (j_outer=0; j_outer<n; j_outer+= vlen-1)
    {
     V0= arrc[j_outer:min(n, j_outer+ vlen-1)][i];       /* hoisted */
     for (k=0; k<n; ++k)
       {
        V1= arra[j_outer:min(n, j_outer+ vlen-1)][k];
        V1= V1* arrb[k][j];
        V0= V0+ V1;
       }
     arrc[j_outer:min(n, j_outer+ vlen-1)][i]= V0;       /* sunk */
    }
```

Inhibitors of Parallelization

Any condition that prevents vectorization of a loop can also prevent parallelization. Specific factors that inhibit or prevent parallelization are:

- Loops with multiple entries or multiple exits
- Loop-carried dependencies (LCDs)
- Function calls within a loop

The following subsections discuss the effect of subroutine calls and dependencies on parallel optimization.

Loop-Carried Dependency

Chapter 3 defines recurrence and dependency and how they affect vectorization. Only backward dependencies interfere with vectorization. Both forward and backward LCDs interfere with parallelization.

The following loop has no dependencies. It can be strip-mined and vectorized by the compiler.

```
for (i= 0; i<n; ++i)
  arr[i] = arr[i] + 3.14;
```

The outer, strip-mine loop is transformed by the compiler so that it can run in parallel on a multiprocessor machine. The result is a *parallel vector loop.*

The following loop, however, has a backward LCD caused by the assignment to `arr[i+1]`. The loop cannot be vectorized or parallelized by the compiler. It remains in scalar form.

```
for (i= 0; i<n; ++i)
  arr[i+1] = arr[i] + 3.14;
```

The next loop has a forward LCD. Because forward LCDs do not interfere with vectorization, the loop is strip-mined and vectorized by the compiler. It is not safe to parallelize a loop that has a forward LCD, however. The result is a strip-mined vector rather than a parallel vector loop.

```
for (i= 0; i<n; ++i)
  arr[i] = arr[i+1] + 3.14;
```

If a loop has dependencies that prevent the compiler from automatically parallelizing it, you can instruct the compiler to parallelize the loop by inserting *synchronization code* to honor the dependencies. This code causes the execution of a thread to halt momentarily until an operation in another thread, on which the halting thread is dependent, has been performed. The overhead caused by synchronization code is often greater than performance gains from parallelization. Synchronized parallel loops are only advantageous if the amount of code within the loop that contains dependencies is relatively small compared to the amount of code that does not contain dependencies. Use the `synch_parallel` directive to instruct the compiler to parallelize and synchronize the executions of a loop.

The `synch_parallel` directive also enables the compiler to parallelize most scalar assignments and reductions within loops. Synchronization enables the compiler to parallelize the following loop. Synchronization ensures that both `s` and `x` end up with the proper values:

```
/* $dir synch_parallel */
for (i=1; i<n; ++i)
  if (arra[i]<0)
    {
     s+= arrb[i] * arrc[i];
     x = arrb[i];
    }
```

Loops with Function Calls

The compiler does not normally parallelize a loop that contains a function call. You can force the compiler to parallelize such a loop by using the `force_parallel` directive. This directive causes the compiler to parallelize the loop that follows it, ignoring any function calls or potential dependencies it detects. Scalar dependencies, which the compiler can conclusively identify as real, are not ignored.

If you use the `force_parallel` directive to parallelize a loop containing a function call, the function called from the loop must be compiled using the `-re` option. The `-re` option causes the compiler to generate reentrant code for a function, using thread-private memory for all local variables. If you do not use `-re`, correct performance of a function called from a parallel loop is not guaranteed.

CAUTION

If a function declares static variables or assigns values to globals, the `-re` option does not ensure reentrancy. Functions that declare static variables or assign to globals are not reentrant and must not be parallelized.

If you use `force_parallel` to parallelize a loop that contains an actual recurrence, the behavior of the loop is nondeterminate. Errors can result at runtime, but no amount of testing can guarantee that an error will be revealed. Before using `force_parallel`, analyze your data and algorithms to ensure that your code can be safely parallelized.

The use of `force_parallel` and `-re` does not guarantee correct performance if your program depends on the order of calls to a reentrant function. Consider the following example:

```
pr_count()
{
 ...
 /* $dir force_parallel */
 for (i = 0; i<100; i++)
   n= getval(i);
 ...
}

getval(n)
int n;
{
 n++;
 printf("%d \n", n);
 return(n);
}
```

The output of this program changes when the optimization level is increased from `-O2` to `-O3`:

-O2	-O3
2	2
3	3
4	4
5	5
6	26
7	6
8	27
...	...
94	98
95	94
96	99
97	95
98	100
99	96
100	97

To parallelize this program without changing the output, you must use `synch_parallel` instead of `force_parallel` and compile for reentrancy using the `-re` option. Because this particular program does not have any significant blocks of code that are free of dependencies, `synch_parallel` does not produce any performance gains, however. This function should be compiled at `-O2` instead.

CONVEX C supports only one level of parallelism. If a function is called from a parallel loop, any parallelized code within that function is executed in serial during that function call. If a function is called only from parallel loops, do not compile that function at optimization level `-O3`. This prevents the compiler from generating dead code that is meant to spawn parallel loops.

Parallelizing Code Outside of Loops

The compiler does not automatically parallelize code outside of a loop. You can use *tasking directives* to instruct the compiler to parallelize this code. The directive `begin_tasks` tells the compiler to begin parallelizing a series of tasks. `next_task` marks the end of one task and the start of the next. `end_tasks` marks the end of a series of tasks to be parallelized.

A section of code containing three tasks would look like this:

```
/*$dir begin_tasks*/
<statement 1>
/*$dir next_task*/
<statement 2>
<statement 3>
/*$dir next_task*/
<statement 4>
/*$dir end_tasks*/
```

The compiler transforms this code into a parallel loop and creates machine code equivalent to

```
/*$dir force_parallel*/
for (i=1; i<4; ++i)
  switch (i)
    {
     1: <statement 1>
        break;
     2: <statement 2>
        <statement 3>
        break;
     3: <statement 4>
        break;
    }
```

One common, effective use of tasking directives is initializing large independent (nonoverlapping) arrays. Tasks should be fairly large (coarse-grained) to minimize the overhead of parallelization and should be of similar size to distribute the work evenly between processors.

Conclusion

The CONVEX C compiler automatically parallelizes a program to exploit the parallel features of CONVEX architecture. No special syntax is required; the compiler accepts standard C programs and generates high-performance executable code.

Chapter 5
Optimizing C Applications

Strategy

For programs that manipulate arrays, vectorization usually offers the greatest possible performance gains. Focus your optimization efforts first on vectorizing those loops and functions that account for the major part of your program's execution time. Once you have obtained the best possible vector performance, you can frequently achieve additional gains through parallelization.

Note	It is very easy to produce a fast program that no longer gives correct results. The goal of optimization is to make a program run fast without affecting results. At each stage of the optimization process, test to make sure that the optimized program still gives correct results.

Step 1 - Compiling a Scalar Program

1) Compile the program at optimization level `-no`:

```
% cc prog.c -no -o no.exe
```

2) Execute the resulting program (`no.exe`) and check the output. If the program is being ported from another machine, compare the new output with previous output. Otherwise, compare the output with expected values. The only differences you find should be those that result from floating-point round-off. If the output does not match expected results, use `csd` to locate and repair the logic error that is causing the problem. If no logic error can be found, use the `contact` utility (described in Appendix B) to report a possible compiler bug.

   It is important that you do not skip this first step. Optimizations performed at higher levels make debugging much more difficult. Be sure your program is giving correct results before you start to add optimization.

Step 2 - Increasing the Optimization Level

1) Recompile your program at optimization level `-O1`. Use the `-pa` option to include instrumentation for profiling under `CXpa`. If you use the older profilers contained in the CONVEX Consultant instead of CXpa, you can still perform most of the steps presented in this chapter. (However, you cannot analyze the performance of individual loops without CXpa.) Refer to the *CONVEX Consultant User's Guide* to determine the appropriate options and commands for using the Consultant.

```
% cc prog.c -pa -O1 -o O1.exe
```

Though possible, it is extremely rare for code to slow down when global optimization (-O1) is applied. If you run into serious difficulties at higher optimization levels, it may be necessary to recompile part of the program at -O0. Barring such difficulties, there is no need to compile at -O0.

2) Time the program using CXpa. Note which functions take the most CPU time, then concentrate your efforts on optimizing these functions. Use the routine-level profile created by CXpa as a baseline against which you can measure the effects of higher optimizations.

```
% cpa O1.exe
CONVEX Performance Analyzer.
Type 'help' for help.
reading executable O1.exe...
(cpa) monitor routine all
(cpa) run > O1.out
(cpa) analyze routine > O1.prof
```

3) Compare the output of your program with the output produced in 2) of Step 1. The only differences should be those that result from the floating-point round-off. Global scalar optimization rarely affects the output of a program. If you do find significant differences in the output between -O0 and -O1 use a binary search to isolate the function responsible for the differences. Compile half the functions at -O0 and half at -O1. Link and execute the program to determine which group contains the offending function. Then, split the suspect group in half. Continue this process until you isolate the offending function.

When you have isolated the function that causes the output to change at the higher optimization level, check the source code for that function and fix any errors you find. If you find no logic errors, recompile that single function at -O0 and proceed to optimize the rest of the program.

Step 3 - Adding Vectorization

You can approach vectorization in two ways. The more common approach is simply to compile the entire program at optimization level `-O2`. Nothing is wrong with this approach, except that it may not be safe or desirable in all cases. If a program has hidden dependencies, misuses directives, or encroaches on the limits of floating-point precision, the code may no longer produce the same output when it has been vectorized. It is also possible, although rare, that code will actually slow down due to vectorization. Reasons for these phenomena are discussed in Chapter 7, "Pitfalls of Optimization."

Step 3a presents an alternative approach. Although safer, this approach can also be more time-consuming. If you have compiled complete programs at `-O2` in the past and achieved good results, there is no reason why you should not continue with that approach. If you experience problems after compiling at `-O2` (your code slows down or gives incorrect answers), you should backtrack and carry out Step 3a; otherwise, go on to Step 4. If you have had problems with vectorization in the past, however, you might want to begin with the procedures outlined in Step 3a.

Do not try to vectorize a function that produces incorrect results at optimization level `-O1`. All scalar optimizations performed at `-O1` are also performed at `-O2` and `-O3`. Any problems caused caused by scalar optimization at `-O1` are sure to recur at `-O2`. If a function gives incorrect results at `-O1`, compile it at the highest optimization level (`-O0` or `-no`) that gives correct results.

Step 3a - Adding Selective Automatic Vectorization

1) Look at the CXpa output from the previous step to determine which function or functions account for a majority of the CPU time. Vectorize the hot functions by compiling with the `-O2` option on the `cc` command line. Compile the rest of the program at `-O1` and link with the vectorized functions:

```
% cc -O2 -c -pa hot_routines.c
% cc -O1 -pa prog.c hot_routines.o -o O2.exe
```

2) Run the new program under CXpa to determine which function now accounts for the greatest CPU time.

3) Compare the output file with `no.out`. Once again, the only differences should be those resulting from floating-point round-off. If you find other differences, recompile some of the hot functions at `-O1` until you have determined the function where the error occurred. Examine this function using `csd` to pinpoint the error. The loop-level output from CXpa may also be helpful in determining, for example, if a particular piece of code never executes. If you cannot find and repair the logic error, recompile the affected function with scalar optimization only (`-O1`).

Repeat 1), 2), and 3) until you have vectorized all functions that account for a significant percentage of CPU time and produce correct results when compiled at `-O2`.

Step 4 - Enhancing Vectorization

1) Using CXpa, produce a loop-level profile of the vectorized functions that use the most CPU time. Ignore any functions compiled at `-O1`; they are fully optimized at this point.

```
% cpa O2.exe
CONVEX Performance Analyzer.
Type 'help' for help.
reading executable O2.exe...
(cpa) monitor loop in hot1, hot2
(cpa) run > O2.out
(cpa) analyze routine > hot.prof
```

2) Examine the optimization report and the CXpa loop profile to determine which loops were vectorized and what transformations were performed. Determine which loops account for the greatest time in the hot routines and work only on these loops.

Your main goal is to increase the number of vectorized loops. Look for apparent recurrences that prevent the compiler from vectorizing loops. Once you have made sure that a recurrence is not real, use the `no_recurrence` directive to tell the compiler it is okay to vectorize the loop.

If your code uses pointers, some loops may not vectorize because of aliasing. Refer to Chapter 7, "Aliasing," for advice on eliminating aliasing problems from your code.

Conditional (`if` or `? :`) statements sometimes prevent vectorization of a loop. If a loop containing a conditional statement does not vectorize, try rewriting the code so that the condition is removed from the loop.

3) When you are satisfied that no more loops can be vectorized, you may still be able to improve the efficiency of your code:

- Simplify conditionals. Vectorized loops that contain conditionals can still be enhanced if the conditional is simplified or removed.
- Simplify array subscripts. A loop with complex subscripts takes longer to execute, even in vector mode, than a loop with simple subscripts.
- Watch for loops with short vector lengths (small trip counts). These loops usually run faster in scalar form. If the trip count of a loop is less than five and the compiler has vectorized the loop, use the `scalar` directive to prevent vectorization.
- Watch for unnecessary strip-mining and inefficient strip-mine lengths. Use CXpa to determine whether a vector loop is strip-mined or not. Refer to Chapter 9 for examples.

For specific examples of how to tune your code for better vectorized performance, see Chapter 8, "Optimization Tricks and Tips."

4) Recompile your modified code at `-O2`.

5) Run this version under CXpa and note the effects of changes you have made. Create an output file called `O2.out`. Check the CPU time for each loop that you have modified. If the time is not improved, remove the modification and recompile.

6) Compare the output file with `no.out`. Once again, the only differences should be those that result from floating-point round-off or calls to timing routines. If significant differences exist between outputs, use `csd` to pinpoint the directive that is causing the problem. Remove the directive and recompile.

Repeat Step 4 until the program produces correct results and loops that represent the majority of CPU time have been vectorized.

Step 5 - Adding Parallelization

At best, parallelization can provide performance improvements in rough proportion to the number of CPUs on your machine. Because parallelization spreads processing across all available CPUs, increasing CPU use, compare turnaround times rather than CPU times. To get accurate times at this stage, you must have exclusive use of your machine.

If your program runs on a machine with multiple CPUs and turnaround time is important, consider parallelizing your program. Otherwise, go on to Step 7. To realize performance gains from parallelization, your program must run on a lightly or moderately loaded machine where there are likely to be CPUs available for parallel execution. If your program runs in a heavily loaded environment, go on to Step 7.

Just as there are two approaches to vectorization, there are two approaches to parallelization. All comments made about vectorization in Step 3 apply to parallelization, except that performance gains due to parallelization are usually smaller than those due to vectorization and the likelihood of running into problems is greater. Use your own experience in deciding whether to compile your entire program at `-O3` or to use the detailed, incremental approach outlined below. As before, if you do have problems using the "all-at-once" approach, revert to the incremental approach.

1) Use the `-O3` option to compile the functions that account for the greatest percentage of turnaround time (wall-clock time) for parallel optimization. Compile the rest of the program at `-O2`.

```
% cc -O3 prog.c -o O3.exe
```

2) Run the program under `/bin/time` in single-user mode and note the turnaround time.

```
% /bin/time O3.exe > O3.out
```

If turnaround time is increased, your program may not benefit from parallelization.

3) Compare the output file with `no.out`. Once again, the only differences should be those that result from floating-point round-off or calls to timing routines. If the output differs significantly, use `csd` to locate the error. Correct the logic error, if it can be found, or recompile the affected function at `-O2`.

4) Recompile the program with the `-pa` option and run it under CXpa. Note the functions that collectively account for the majority of wall-clock time.

If a function appears to run slower (in terms of wall-clock time) at `-O3` than it did at `-O2`, recompile that function at `-O2`. If any remaining function appears to account for a significant amount of wall-clock time, recompile that function at `-O3`.

Repeat Step 5 until the program produces correct results and the functions that account for a majority of turnaround time have been parallelized. With the version that produces correct results in the least time, proceed to the next step.

Step 6 - Enhancing Parallelization

1) Examine optimization reports and CXpa output for each parallelized function. Note which scalar loops failed to parallelize. Any scalar loop that uses significant wall-clock time may be a candidate for parallelization. Loops that are less deeply nested derive greater benefit from parallelization.

```
% cpa O3.exe
CONVEX Performance Analyzer.
Type 'help' for help.
reading executable O3.exe...
(cpa) monitor loop in hot_routine1 hot_routine2...
(cpa) run > O3.out
(cpa) analyze loop
```

2) Find the the real apparent dependencies that prevent the compiler from parallelizing a scalar loop. Remove an apparent dependency by applying the appropriate directive to the loop (`no_recurrence` or `force_parallel`). Take care to avoid using these directives on loops containing real recurrences.

3) Compile modified functions at `-O3`. Compile the rest of the program at `-O2` or `-O3`, as before.

```
% cc -O1 prog.c -o O3.exe
```

4) Run the program under `/bin/time` in single-user mode and note the turnaround time.

```
% /bin/time O3.exe > O3.out
```

If the turnaround time is not improved, abandon the modification.

5) Compare the output file with `no.out`.

```
% diff O3.out no.out
```

Once again, the only differences should be those that result from floating-point round-off or calls to timing routines. If the output differs significantly, use `csd` to locate the directive that is causing the problem. Remove the directive and recompile.

6) Recompile the program using the `-pa` option. Run it under CXpa. Note which functions take the greatest turnaround time.

```
% cc -O1 -pa prog.c -o O3.exe
```

Repeat Step 6 until the program produces correct results and the loops that account a majority of turnaround time have been parallelized.

Step 7 - Wrapping Up

The `-pa` option causes the compiler to insert timing code and data, also called instrumentation, into your program. When the program is fully optimized, recompile without the `-pa` option to remove the instrumentation overhead.

Conclusion

You can often enhance the compiler's ability to optimize the program by modifying the algorithm or applying directives. To do so safely, you must know the code and data. You must also make sure that code modifications do not affect the output of the program.

Chapter 6
Efficient Program Constructs

The optimization techniques used by the CONVEX C compiler were developed originally for optimizing FORTRAN programs. C is a richer language than FORTRAN, with a wider range of programming constructs. Some of these programming constructs can make optimization difficult or impossible. Selecting the proper C programming constructs can make your program much easier to optimize. In general, the most efficient C programming constructs, and the easiest to optimize, are those that most closely parallel FORTRAN constructs.

Float versus Double

Floating-point variables in C may be of type `float` or type `double`; unless otherwise specified, floating-point constants are of type `double`. Data of type `float` are stored as a 32-bit floating point number, while data of type `double` are stored as a 64-bit number. For this reason, you might expect operations on type `float` to be faster than operations on type `double`. The original C language definition specifies that all operations on type `float` are to be done using type `double`, however, and CONVEX C follows this convention in non-ANSI (backward-compatible) mode. The overhead involved in converting `float` operands to type `double` and converting the result from `double` back to `float` makes operations on type `float` slower than operations on type `double`.

Programs that define all floating-point numbers to be of type `double` do not incur the overhead of type conversion. This is not generally acceptable, however, and the result is still not as fast as true single-precision arithmetic. A better alternative is the `-fd` option, which is the default when a program is compiled in ANSI mode.

The `-fd` compiler option causes all operations on type `float` to be performed in single-precision (using 32-bit floating-point numbers). In non-ANSI (backward-compatible) mode, scalar variables of type `float` are still converted to type `double` before being passed to functions. In ANSI mode, they are converted only if no prototype for the function exists. Floating-point constants are still type `double` by default. The `-float sp_const` option causes floating-point constants to default to type `float`.

Mixed-Mode Operations

Integer operations are usually faster than floating-point operations. For vector operations, the difference is usually quite small. However, when integer and floating-point operations are combined in the same expression, the overhead caused by type conversions can be significant. Avoid writing mixed-mode expressions, especially within vectorized loops.

Writing Efficient Loops

C provides many options for writing loops. The only loop that the compiler can vectorize, however, is a *counted loop*. A counted loop is a loop whose iteration value can be determined before the loop is executed. This iteration count is required to determine the number and length of the vector strips.

A counted loop must have an induction variable (a variable that is incremented or decremented by a fixed amount on every iteration) and a fixed stop value.

The most common type of counted loop is a simple `for` loop:

```
for (i=0; i<1000; i++)
  arra[i]= arra[i]*arrb[i];

for (i=1000; i>0; i--)
  arra[i]= arrb[i]/4.16;

for (i=0; i<n; i++)
  arra[i]=arra[i]* 4.16;
```

In these three examples, the induction variable is `i`. Note that `i` is assigned at the started of the loop, incremented or decremented by a constant amount on every iteration, and tested in the head or control section of the loop on every iteration. The induction variable must be used in an array subscript within the body of the loop. Otherwise, the code is loop-invariant; the compiler moves the invariant code out of the loop and eliminates the loop instead of vectorizing it.

If a loop does not increment or decrement the iteration variable by a constant integer amount, the compiler cannot find an induction variable and cannot vectorize the loop. The lack of an induction variable prevents the compiler from vectorizing the following loop:

```
for (i=1; i<1000; i*=2)
  arra[i]=arra[i]* arrb[i]* arrc[i];
```

In this loop, `i` takes the value of 1, 2, 4, 8, 16, and 32 on successive iterations. Because `i` is incremented by a different amount on each iteration, it is not an induction variable, and the compiler cannot vectorize the loop.

This loop can be unrolled by hand:

```
a[1]= b[1]* c[1];
a[2]= b[2]* c[2];
a[4]= b[4]* c[4];
a[8]= b[8]* c[8];
a[16]= b[16]* c[16];
a[32]= b[32]* c[32];
a[64]= b[64]* c[64];
a[128]= b[128]* c[128];
a[256]= b[256]* c[256];
a[512]= b[512]* c[512];
```

The unrolled version of this loop runs approximately 120 percent faster than the original version.

Sometimes, code may be written in such a way that it hides the induction variable from the compiler. The following code uses `i` as an implicit induction variable, but the compiler cannot recognize this and cannot vectorize the loop:

```
i=0;

for ( ; ;)
  {
   arrb[i]= arra[i]* 4.16;
   arra[i]= arrc[i];
   if (i++ > n)
     break;
   arrb[i]= arrb[i]- 1.0;
   arra[i]= arra[i]/ arrb[i];
  }
```

Rewrite the code as follows:

```
for (i=0; i<=n; i++)
  {
   arrb[i]= arra[i]* 4.16;
   arra[i]= arrc[i];
  }

for (i=0; i<n; i++)
  {
   arrb[i]= arrb[i]- 1.0;
   arra[i]= arra[i]/ arrb[i];
  }
```

Both of these loops now have explicit induction variables and can be vectorized by the compiler.

If the iteration variable of a `for` loop is incremented by a non-integer constant, the compiler cannot find an induction variable and the loop cannot vectorize:

```
for (i=0; i<n; i+=2.0)
  arra[i]=arra[i]* 4.16;
```

CAUTION If a loop start, stop, or iteration value falls outside the range of `int` (that is, it cannot be stored in 32 bits) the compiler may truncate the value to 32 bits when it vectorizes the loop. Avoid using start, stop, or iteration values that exceed the range of `int`.

The second expression in the head of a `for` loop is the conditional test. The test can use any of the following comparison operators: >, <, <=, >=, or !=.

```
for (i=0; i<n; i++)
  arra[i]= arrb[i];

for (i=1000; i>n; i--)
  arra[i]= arrb[i];

for (i=0; i<=n; i++)
  arra[i]= arrb[i];

for (i=1000; i>=n; i--)
  arra[i]= arrb[i];
```

The conditional test compares the induction variable to a *stop value*. The stop value can be a variable or a constant, but its value must not change within the loop. The following loop cannot be vectorized because the stop value can change within the loop:

```
for (i=0; i<n; ++i)
  {
   iarra[i]= iarrb[i];
   if (iarra[i+1]>0)
     n= iarra[i+1];
  }
```

If the array `iarra` contains a positive value within the range of 0 to `n`, the value of `n` is altered. The compiler does not know what value `n` might take or what the contents of `iarra` will be at runtime. It cannot, therefore, ensure that `n` is loop invariant. It must treat this loop as an uncounted loop, which cannot be vectorized.

Loops with more complicated conditional tests, such as the following example, generally do not vectorize:

```
for (i=0, j=0; i<n && j<n; i++, j+=m)
  arra[i]= arra[i+j];
```

Loops that use pointers or array references as control variables may not vectorize because of aliasing:

```
for (i=0; i< *n; i++)
  arra[i]= arra[i]+ arrb[i];

for (i=0; i< n[0]; i++)
  arra[i]= arra[i]+ arrb[i];

for (i=0; i< n[0]; i+= *p)
  arra[i]= arra[i]+ arrb[i];
```

See Chapter 7 for more information on aliasing.

If a loop has more than one exit, the compiler cannot tell what part of the code in the body of the loop will execute at runtime. As a result, loops that contain alternate exits, such as the following example, do not vectorize:

```
for (i=0; i<n; i++)
  {
   a[i]=c[i]+ d[i];
   if (a[i]< 7.0) break;
   a[i]=a[i]/2.0;
  }
```

A `for` loop may iterate on more than one variable. Such loops can vectorize, as long as all other restrictions are met.

```
for (i=0, j=1000; i<n; i++, j+=2)
  arra[i]= arrb[j];
```

Most loops that contain conditionals (`if` tests or `?:` operators) can be vectorized. Vector loops that contain conditionals are less efficient than vector loops that do not contain conditionals. Remove conditionals from loops wherever possible. Check boundary conditions before or after rather than within the loop.

The following code shows a series of conditionals imbedded within a `for` loop. In this case, the conditionals do not prevent vectorization of the loop, but do make the vectorized loop slower:

```
for (i=0; i<10000; i++)
  {if (i<2000)
     {
      arrc[i]= arra[i]* 2000.0 + cos(arra[i]);
      arrb[i]= arrb[i]* arrc[i]*arrc[i]*arrc[i]*arrc[i]/ arra[i];
     }
   if ((i>2000) && (i<4000))
     {
      arrc[i]= arra[i]+ cos(arra[i]);
      arrb[i]= arrb[i]+ arrc[i]
     }
   if ((i>4000) && (i<6000))
     {
      arrc[i]= arra[i]+ 2000.0;
      arrb[i]= arrb[i]* arrb[i]* arrb[i]* arrb[i];
     }
   if (i>6000)
     {
      arrc[i]= arra[i];
      arrb[i]= 1.0;
     }
  }
```

Remove the conditional by splitting the single `for` loop into four separate loops:

```
for (i=0; i<2000; i++)
  {
   arrc[i]= arra[i]* 2000.0 + cos(arra[i]);
   arrb[i]= arrb[i]* arrc[i]*arrc[i]*arrc[i]/ arra[i];
  }

for (i=2000; i<4000; i++)
  {
   arrc[i]= arra[i]+ cos(arra[i]);
   arrb[i]= arrb[i]+ arrc[i];
  }

for (i=4000; i<6000; i++)
  {
   arrc[i]= arra[i]+ 2000.0;
   arrb[i]= arrb[i]* arrb[i]* arrb[i]* arrb[i];
  }

for (i=6000; i<10000; i++)
  {
   arrc[i]= arra[i];
   arrb[i]= 1.0;
  }
```

This simple manipulation improves the code's performance by approximately a factor of 10.

The following code shows an example of boundary tests that can be removed from a loop:

```
for (j=0; j<1000; j++)
  for (i=0; i<1000; i++)
    {
     if ((i==0)||(i==999))
       {
        if ((j==0)||(j==999))
          arra[i][j]= 0.0;
        else
          arra[i][j]= arrb[i][j];
       }
     else
       arra[i][j]= arrb[i][j];
    }
```

When the boundary values are set outside the loop, this code fragment runs about five times faster:

```
for (j=0; j<1000; j++)
  for (i=0; i<1000; i++)
    arra[i][j]= arrb[i][j];

arra[0][0]=     0.0;
arra[0][999]=   0.0;
arra[999][0]=   0.0;
arra[999][999]= 0.0;
```

A `while` loop is similar to a `for` loop. Like a `for` loop, a `while` loop can be vectorized if it is a counted loop. The following example shows a counted `while` loop, which can be vectorized:

```
i=0;
while (i< n)
  {
   a[i]= b[i]+ c[i];
   ++i;
  }
```

The induction variable in this loop is `i`. It is set prior to the start of the loop, incremented within the body of the loop, and tested on each iteration. Another form of this loop increments the induction variable within the head instead of the body:

```
i=0;
while ((i++)< n)
  {
   a[i]= b[i]+ c[i];
  }
```

In both examples, `n` is loop invariant. Both loops have recognizable induction variables and fixed stop values. Both loops are vectorized by the compiler. Most hand-coded loops (loops that are written using `goto` statements) and `do-while` loops are uncounted and are not vectorized by the compiler. If a hand-coded loop or `do-while` loop has a fixed stop value and recognizable induction variable, it can be vectorized. The following example shows a `do-while` loop that can be vectorized:

```
i= 0;
do
  {
   a[i]=b[i]* c[i];
  }
while (++i<n);
```

Because most hand-coded loops and `do-while` loops do not vectorize, avoid writing code that depends heavily on these constructs.

Multidimensional Arrays

Most C programming courses and C programming textbooks teach students to write programs that rely on arrays of pointers instead of multidimensional arrays. The rationale for using this technique is that operations on arrays of pointers are faster than operations on multidimensional arrays. On many machines, perhaps even most machines, this is true, but on the CONVEX C Series supercomputer, using multidimensional arrays is faster.

The following example shows a "textbook" matrix addition, using arrays of pointers.

```
mat_add_ptrs(m1, m2, r, n)
register double *m1[], *m2[], *r[];
register int n;
{
 register int i, j, k;

 for (i=0; i<n; i++)
   /* $dir no_recurrence */
   for (j=0; j<n; j++)
     r[i][j]= m1[i][j] + m2[i][j];
}
```

Rewritten to use multidimensional arrays, the function looks like this:

```
#define N 100

mat_add_arrys(m1, m2, r, n)
double m1[N][N], m2[N][N], r[N][N];
register int n;
{
 register int i, j, k;

 for (i=0; i<n; i++)
   for (j=0; j<n; j++)
     r[i][j]= m1[i][k] + m2[k][j];
}
```

The CPU has to do less work to calculate the memory addresses for the multidimensional arrays used in the second example. As a result, the second example runs at least 20 percent faster in scalar form (that is to say, when it is compiled at `-O1` or below).

When the loops in this function are vectorized, the difference becomes much more dramatic. Typically, vector loops that access multidimensional arrays run 5 to 10 times faster than vector loops that access arrays of pointers. The reasons for this dramatic improvement are:

- Multidimensional arrays occupy contiguous memory locations. As a result, the CPU must do less work to calculate effective addresses.
- Multidimensional arrays can sometimes be processed as a single vector strip. Pointers in an array must be processed one at a time, creating an extra level of strip-mining.

Pointers can also cause aliasing problems that prevent vectorization and slow down the execution of your code. Refer to Chapter 7, "Aliasing," for a discussion of the problems caused by pointers.

Optimizing Array Accesses

In C, arrays are stored in row-wise order. As a result, accessing the trailing or right-most dimension of an array is faster than accessing the leading or left-most dimension. Write your arrays so that most of the accesses (assignments and references) are made to the trailing or right-most dimension.

The following example shows three loops, in order of increasing efficiency (speed):

```
for (i=0; i<n; i++)            /* least efficient */
  arra[i][0][0]= 4.0;

for (i=0; i<n; i++)
  arra[0][i][0]= 4.0;

for (i=0; i<n; i++)            /* most efficient */
  arra[0][0][i]= 4.0;
```

CONVEX C automatically interchanges many loops to optimize the efficiency of array accesses. In the following example, the compiler interchanges the `j` and `i` loops:

```
for (j=0; j<n; j++)
  for (i=0; i<n; i++)
    arra[i][j][0]= 4.0;
```

This produces

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    arra[i][j][0]= 4.0;
```

If the trip count of an outer loop is much smaller than the trip count of the inner loop, the compiler does not interchange the loops even though it could achieve more efficient memory accesses by doing so. The compiler realizes that a few slow memory accesses can be faster than many faster accesses. In other cases, where the compiler cannot determine the trip count, it may interchange two loops to achieve fast memory accesses even though this results in a much larger average trip count on the outer loop. If you write most of your loops to access the trailing dimension of an array, you can minimize the number of compromises the compiler must make.

Conclusion

An understanding of the way the compiler operates and the rich set of programming constructs provided by the C language allows you to select those constructs that can be optimized most efficiently. In general, the most efficient C programming constructs, and the easiest to optimize, are those that most closely parallel FORTRAN constructs.

Chapter 7
Aliasing

Aliasing occurs when two or more variable names point to the same memory location. A potential alias can prevent the compiler from vectorizing some code. In some cases, the `-alias array_args` option and the `no_recurrence` directive may be used to allow the vectorization of code that would not otherwise be vectorized. Care must be taken to ensure that this does not change the output of the program.

Aliasing and Dependency

An *alias* is an alternate name for some object. Aliasing occurs in a program when two or more names are attached to the same variable or location in memory. A *potential alias* occurs when the compiler cannot tell whether two names are attached to the same memory location.

Aliasing compounds the problems of dependency and recurrence. Consider the following example:

```
for (i=0; i<n; ++i)
  arra[i]= arrb[i];
```

This loop appears to be free of backward dependencies. Assuming `arra` and `arrb` are distinct, it can be safely vectorized. However, if `arra[i]` is an alias for `arrb[i]` (that is, if `arra` and `arrb` overlap), a backward dependency may exist. If `arra` and `arrb` are aliased together, the code is equivalent to

```
for (i=0; i<n; ++i)
  arrb[i+K]= arrb[i;
```

where `K` is a value that determines how the arrays overlap. If `K` is positive, a backward dependency exists. When aliasing occurs, this value cannot be determined at compile time, so the compiler cannot know whether a backward dependency exists. To avoid possible undetected dependencies, the compiler does not attempt to vectorize this loop.

Why Aliasing Occurs

The C language supports explicit aliasing in the form of `unions`. This type of aliasing rarely causes problems in vectorization. More often, vectorization problems are caused by implicit aliasing that results from the use of pointers. Pointer aliasing, in fact, is one of the most frequent causes of optimization difficulties in CONVEX C.

Consider the following example. Function `alex` has three parameters, `a`, `b` and `n`. `a` and `b` are pointers of type `float`. `n` is an `int`.

```
alex(a, b, n)
float *a;
float *b;
int n;
{
 int i;

 for (i=0; i< n; i++)
   a[i]=  b[i]/2.0;
}
```

Pointers `a` and `b` point to array addresses that are passed at runtime. Depending on the addresses passed to the function, the array pointed to by `a` may overlap the array pointed to by `b`. The possibility of overlap creates a potential alias. Because of this potential alias, the compiler assumes a dependency may exist and does not vectorize the loop.

This loop is vectorized if you use a `no_recurrence` directive. Unless you know that the arrays pointed to by `a` and `b` do not overlap (aliasing does not occur), this is an unsafe use of the `no_recurrence` directive.

Aliasing Algorithms

The algorithm the compiler uses to determine whether a potential alias exists depends on the compilation mode. The algorithm used by CONVEX C compilers prior to V4.0, called *worst-case aliasing*, is still used when a program is compiled in backward-compatible (non-ANSI) mode. When a program is compiled in an ANSI mode, the new, ANSI aliasing algorithm is used.

The *worst-case aliasing* views all pointer references as subscripts into a giant array that encompasses all of memory. This mythical array (known as `*MEM*` in compiler optimization reports) includes all global variables, local variables whose address is taken using the address (`&`) operator, and local static variables. (This treatment of static variables is necessary to allow for recursion, since a static is, in a sense, "global" to a function on a recursive call.) As a result:

- Every pointer's reference is a potential alias for every other pointer's reference.
- Every pointer's reference is a potential alias for every global variable (and vice—versa).
- Every pointer's reference is a potential alias for every local variable whose address is taken using the `&` operator.
- Every pointer's reference is a potential alias for every local static variable.

The ANSI algorithm is somewhat different. ANSI C provides stricter type-checking, which allows the compiler to use a stricter algorithm that finds fewer potential aliases. Instead of one giant array, the ANSI algorithm assumes a separate array for each base type such as `int`, `float`, or `double`. Pointers and variables can only be aliased with pointers and variables of the same base type. Aliasing algorithms are applied on a function-wide basis, before optimization takes place.

The following example illustrates the difference between the two aliasing algorithms:

```
alex1(a, ib, n)
float *a;
int *ib;
int n;
{
 int i;

 for (i=0; i< n; i++)
   a[i]=  ib[i]/2.0;
}
```

Here, `a` and `ib` are pointers to different base types. Therefore, under ANSI C rules, no potential alias exists, and the loop is vectorized. Under the old worst-case aliasing rules, a potential alias does exist and the loop is not vectorized.

The ANSI aliasing algorithm may not be safe in all cases, especially if your program uses illegal code. The following code fragment shows how pointers of different base type can become aliased to one another:

```
int array[100];
int *dummy;
float *fptr;

*dummy= array;
fptr= dummy;              /* illegal pointer/integer combination */
```

The assignment to `fptr` makes the code non-ANSI compliant. The code compiles with a warning message (shown as comments in this example). Because the code violates ANSI rules, it is no longer safe to use the ANSI aliasing algorithm. You can use the old, worst-case aliasing algorithm, even when compiling in ANSI mode, by specifying `-alias worst` on the `cc` command line.

In the following code, function foo2 passes two `int` pointers to function foo, which expects a pointer of type `int` and a pointer of type `char`. This is not flagged by the compiler as an error. Because the formal parameters are of different base types, the ANSI aliasing algorithm finds no potential aliases and the compiler vectorizes the loop, even though aliasing exists:

```
foo(ia, cb)
int *ia;
char *cb;
{
 int i;

 for (i=0; i<5000; ++i)
   cb[i]= ia[i];
}

foo2
{
 int arra[5000];
 ...
 foo(&arra[0], &arra[1000]);
}
```

Array Subscripts

It is possible for a variable that appears in an array subscript to become aliased with a pointer. In the following example, the variable `k` is a potential alias for `*iptr` because its address is passed to the function getval. The variable `k` may occupy the same memory address as `iptr[j]` on some iteration of the `j` loop. If this happens, the value of `k` is changed for all subsequent iterations. Because the value of `k` can be changed by an assignment to `iptr[j]`, a real dependency exists. The compiler does not vectorize the loop.

```
subex(iptr, n)
int *iptr, n;
{
 int j, k;

 n= getval(&k,n);
 k=12;

 for (j=0; j<n; j++)
   iptr[j]+= k;
}
```

It is not safe to use the `no_recurrence` directive on this loop.

Induction and Stop Variables

Even when a dependency does not exist, aliasing can stop vectorization by preventing the compiler from recognizing a loop induction variable. In the following example, the address of `j` is referenced using the address operator and assigned to `*ip`. As a result, the aliasing algorithm assumes that `j` is part of the global memory array and, therefore, a potential alias for `*iptr`. Because values are assigned to `iptr[j]` within the loop, this potential aliasing means that the value of `j` may be altered. As a result, `j` is not an induction variable and the `j` loop is not a counted loop that can be vectorized.

```
ialex(iptr)
float *iptr;
{
 int j;
 int *ip= &j;

 for (j=0; j<2048; ++j)
   iptr[j]= 107;
}
```

A `no_recurrence` directive does not solve the problem, because the problem is not caused by dependencies. The solution is to remove the line of code that references the address of `j`.

The use of a pointer as a loop counter has the same effect. The following code is compiled under the worst-case aliasing rules. In this example, the compiler recognizes that `*j` is not an induction variable -- because it is a potential reference for `fptr[*j]`, it can might be altered within the loop -- and does not vectorize the loop:

```
ialex2(fptr, j, n)
float *fptr;
int *j, n;
{
 for (*j=0; *j<n; *j++)
   fptr[*j]= 1.7;
}
```

Because `*j` is type `int` and `*fptr` is type `float`, this loop does vectorize under ANSI aliasing rules.

The aliasing of a stop variable can also prevent vectorization of a loop. In the following example, the stop variable in the `i` loop (`n`) becomes aliased when its address is passed to the function foo. Because `n` is a potential alias for `dptr[]`, its value could potentially be altered within the loop. The `i` loop, therefore, is not a counted loop and cannot be vectorized.

```
salex(fptr, dptr, s, r, n)
float *fptr, *s, *r;
double *dptr;
int n;
{
 int i;

 for (i=0; i<n; ++i)
   dptr[i]= *s * dptr[i]- *r * fptr[i];

 foo(&n, s, r);
 return;
}
```

Note that `n` and `dptr` are of different base types. Under ANSI aliasing rules this loop can be vectorized as written. To vectorize this loop under the worst-case aliasing rules, create a temporary variable to hold the stop value for the loop:

```
salex(fptr, dptr, s, r, n)
float *fptr, *s, *r;
double *dptr;
int n;
{
 int i, tmp;

 tmp=n;
 for (i=0; i<tmp; ++i)
   dptr[i]= *s * dptr[i]- *r * fptr[i];

 foo(&n, s, r);
 return;
}
```

Because `tmp` is not aliased to `dptr[]`, the stop value of the loop is now fixed, and the loop vectorizes.

Global Variables

The potential aliasing of a global variable and a pointer frequently causes optimization difficulties. The following code, for example, uses a global variable as a stop variable. This variable (`n`) is of the same type as the pointer `ik` and must therefore be considered potential aliases. Because the value of `n` could be altered by an assignment to its alias, `ik[]`, the loop in the code is not a counted loop and cannot be vectorized.

```
int n, *ik;
foo()
{
 int i;
 /*$dir no_recurrence */
 for (i=0; i<n; ++i)
   ik[i]=i;
```

To vectorize this loop, use a local variable instead:

```
int *ik;
foo()
{
 int i, n;
 /*$dir no_recurrence */
 for (i=0; i<n; ++i)
   ik[i]=i;
```

Changing the declaration of `ik` from a pointer to a global array also prevents aliasing and allows the loop to be vectorized:

```
int n, ik[1000];
foo()
{
 int i;
 /*$dir no_recurrence */
 for (i=0; i<n; ++i)
   ik[i]=i;



...
subex2(iptr, n)
int *iptr, n;
{
 int j;
 k=12;

 for (j=0; j<n; j++)
   iptr[j] = k;
}
```

Array Parameters

When an array is passed as a parameter to a C function, it is passed as a pointer. This might appear to be a serious problem when you are trying to avoid aliasing. The CONVEX C compiler provides an option, however, that allows the compiler to treat array parameters as arrays, which are not aliased, instead of as pointers. This option is `-alias array_args`.

The loop in the following code does not vectorize because of potential aliasing. Under the ANSI C rules, `arra` is a potential alias for `arrb`. Under the old, worst-case rules, `arra` and `arrb` are also potential aliases for `n`:

```
parex(arra, arrb, n)
float arra[], arrb[];
int *n;

{
  int i;

  for (i=0; i< *n; i++)
    arra[i]=  arrb[i]+ i;
}
```

To vectorize this loop, compile the function with `-alias array_args`. If you are compiling in the backward-compatible mode, or compiling in an ANSI mode using the `-alias worst` option, you must also add a temporary variable to hold the loop's stop value because of the aliasing of `n`:

```
parex(arra, arrb, n)
float arra[], arrb[];
int *n;

{
  int i, stop;

  stop= *n;
  for (i=0; i< stop; i++)
    arra[i]=  arrb[i]+ i;
}
```

The loop now vectorizes.

> **CAUTION** The `-alias array_args` option does not prevent aliasing of array parameters. When you use `-alias array_args`, the compiler assumes that all array parameters are distinct. If you pass overlapping arrays as parameters, this `-alias array_args` option hides the aliasing from the compiler. Undetected dependencies and errors can result.

ANSI C declares that the ability of two array parameters to point to the same object in memory (that is to become aliased to one another), is an obsolescent feature. To ensure compatability with future versions of ANSI C, do not write code that depends on the ability to pass overlapping or aliased arrays to a function.

Preventing Aliases

To prevent aliasing problems, avoid the use of pointers, structure pointers, and address operators whenever possible. Avoid code such as

```
divarrays()
{
 double *a, *b;
 int i;
 ...

 for (i=0; i<1000; ++i)                /* DOES NOT VECTORIZE */
   a[i]= a[i]/ b[i];
}
```

Instead, write

```
divarrays()
{
 double arra[1000], arrb[1000];
 int i;
 ...

 for (i=0; i<1000; ++i)                /* VECTORIZES */
   arra[i]= arra[i]/ arrb[i];
}
```

or use a `no_recurrence` directive:

```
divarrays()
{
 double *a, *b;
 int i;
 ...

 /* $dir no_recurrence */
 for (i=0; i<1000; ++i)                /* VECTORIZES */
   a[i]= a[i]/ b[i];
}
```

The following code does not vectorize because of pointers `arra` and `arrb` and the variable `n`, which is used with the address operator:

```
alex4(arra, arrb, n)
float arra[], arrb[];
int n;

{
  int i;

  for (i=0; i< n; i++)
    arra[i]=  arrb[i]+ i;
  sub1(&n);
}
```

The no_recurrence directive tells the compiler to ignore the potential recurrence caused by the possible aliasing of *arra and *arrb. The variable n still poses a problem, however. Because it is used with the address operator, n may be aliased with *arra and *arrb. This potential alias prevents vectorization. To solve this problem, create a temporary variable m to replace the global n within the loop:

```
alex4(arra, arrb, n)
float *a, *b;
int n;

{
  int i;

  m= n;
  /* $dir no_recurrence */
  for (i=0; i< m; i++)
    a[i]=  b[i]+ i;
  sub1(&n);
}
```

An alternative is to replace pointers a and b with arrays arra and arrb:

```
alex4(arra, arrb, n)
float arra[], arrb[];
int n;

{
  int i;

  for (i=0; i< n; i++)
    arra[i]=  arrb[i]+ i;
  sub1(&n);
}
```

Conclusion

Aliasing, the assignment of alternate names to the same object, prevents the compiler from safely optimizing code. Aliasing can occur any time you use the pointer (*), address (&), or structure pointer (->) operators. When a potential alias exists, the compiler does not automatically vectorize or parallelize the affected code. Affected code can be vectorized or parallelized using no_recurrence directives. (Take care to avoid actual recurrences). To prevent aliasing problems, avoid the use of pointers wherever possible.

Chapter 8
Optimization Tricks and Tips

No matter how sophisticated the compiler is, optimization remains more an art than a science. This chapter presents a number of optimization tricks that have been successfully applied to programs written in CONVEX C, explains underlying principles, and offers tips on how you can apply them in your own C programs.

Eliminate Unnecessary Strip Mines

If the compiler determines that the iteration or trip count of a loop is less than or equal to 128, it does not strip-mine the loop. Loops are often written with variable trip counts. Unless the compiler can determine the value of the trip-count variable (through constant propagation, for example) it must strip-mine the loop to allow for a possible trip count greater than 128. The following code shows a loop with a trip count that varies between 1 and 50:

```
n= getval(n);           /* returns a value from 1 to 50 */
for (i=0; i<n; i++)
  a[i]= b[i]*c[i];
```

In this case, strip-mining produces unnecessary overhead. Because you know that getval never returns a value greater than 50, you can use the `max_trips` directive to prevent strip-mining of the loop:

```
n= getval(n);
/* $dir max_trips(50) */
for (i=0; i<n; i++)
  a[i]= b[i]*c[i];
```

Any value of `max_trips` up to 128 prevents the compiler from strip-mining a loop. Because you know the trip count cannot exceed 50, however, use that value. This permits the compiler to generate a more efficient loop.

Do Not Vectorize Loops with Small Trip Counts

Look for loops with small trip counts. A loop with a trip count less than five is usually not worth vectorizing, and the compiler does not vectorize any loop with a trip count less than three. For loops with variable trip counts or trip counts between three and five, you can use the `scalar` directive to prevent vectorization:

```
/* $dir scalar */
for (i=0; i<4; i++)
  a[i]= b[i]*c[i];
```

Loops with variable trip counts are usually vectorized and strip-mined. The compiler strip mines the following loop because it cannot determine the trip count:

```
n= getval(n);          /* returns a value of 1, 2, 4, 8, 16, or 32 */
for (i=0; i<n; i++)
  a[i]= b[i]*c[i];
```

You can use the `max_trips` directive to prevent this loop from being strip-mined, but half the time this loop has a trip count so small that it should not be vectorized at all. You can distribute this loop and use the `scalar` directive to eliminate the overhead of a vectorized loop when the trip count is less than or equal to four:

```
n= getval(n);
if (n>4)
  /* $dir max_trips(32) */
  for (i=0; i<n; i++)
    a[i]= b[i]*c[i]
else
  /* $dir scalar */
  for (i=0; i<n; i++)
    a[i]= b[i]*c[i];
```

Instead of distributing the loop by hand and using the `scalar` directive, you can use the `select` directive, which tells the compiler to create multiple versions of the loop:

```
n= getval(n);
  /* $dir select(4, *, *) */
  for (i=0; i<n; i++)
    a[i]= b[i]*c[i];
```

This directive tells the compiler to create multiple versions of the loop, one of which is selected at runtime. The parameters say that the vectorized version should be selected if the trip count is greater than or equal to four. The asterisks tell the compiler not to create parallel and vector-parallel versions of the loop.

Because the `select` directives does not require rewriting any code, this approach is usually safer and easier for the programmer. In this case, however, you lose the benefit of the `max_trips` directive, which is generally ignored by the compiler when used with a `select` directive.

Scalar loops with small trip counts may be more efficient if they are *unrolled*. Unrolling replaces a loop with a linear sequence of statements. The following example shows a loop

```
/* $dir unroll */
for (i=0; i<3; ++i)
  a[i]= a[i]+1;
```

that is unrolled into

```
a[0]= a[0]+1;
a[1]= a[1]+1;
a[2]= a[2]+1;
```

The `unroll` directive works only if the compiler can determine that the trip count is less than five. If the loop has a variable trip count, it can be distributed and unrolled by hand.

Promote an Array

Sometimes it is necessary to promote an array to a higher dimension to vectorize a loop. In the following example, only the `j` loop vectorizes. The compiler is unable to vectorize the `i` loop because of a recurrence. Values assigned to `arrq` within the `j` loop depend on values assigned to `arrb` by the four preceding statements. Those values of `arrb` exist only until the next iteration of the `i` loop. (There is a cycle of antidependencies between assignments to `arrb[n]`). This prevents the compiler from distributing the `i` loop.

```
prom(arrq)
double arrq[];

{
 int i, j;
 double arrb[4], arrp[4];

 for (i = 0; i<62500; ++i)       /* SCALAR */
   {
    arrb[0] = gl_arrs[i  ]*arrp[0] + gl_arrs[i+4]*arrp[1] +
              gl_arrs[i+7]*arrp[2] + gl_arrs[i+9]*arrp[3];
    arrb[1] = gl_arrs[i+4]*arrp[0] + gl_arrs[i+1]*arrp[1] +
              gl_arrs[i+5]*arrp[2] + gl_arrs[i+8]*arrp[3];
    arrb[2] = gl_arrs[i+7]*arrp[0] + gl_arrs[i+5]*arrp[1] +
              gl_arrs[i+2]*arrp[2] + gl_arrs[i+6]*arrp[3];
    arrb[3] = gl_arrs[i+9]*arrp[0] + gl_arrs[i+8]*arrp[1] +
              gl_arrs[i+6]*arrp[2] + gl_arrs[i+3]*arrp[3];

    for (j = 0; j<4; ++j)          /* VECTOR */
      arrq[j] += arrb[j];
   }
}
```

To eliminate the recurrence, promote `arrb` to a two-dimensional array:

```
prom(arrq)
double arrq[];

{
 int i, j;
 double arrb[4][62500], arrp[4];

 for (i = 0; i<62500; ++i)
   {
    arrb[0][i] = gl_arrs[i  ]*arrp[0] + gl_arrs[i+4]*arrp[1] +
                 gl_arrs[i+7]*arrp[2] + gl_arrs[i+9]*arrp[3];
    arrb[1][i] = gl_arrs[i+4]*arrp[0] + gl_arrs[i+1]*arrp[1] +
                 gl_arrs[i+5]*arrp[2] + gl_arrs[i+8]*arrp[3];
    arrb[2][i] = gl_arrs[i+7]*arrp[0] + gl_arrs[i+5]*arrp[1] +
                 gl_arrs[i+2]*arrp[2] + gl_arrs[i+6]*arrp[3];
    arrb[3][i] = gl_arrs[i+9]*arrp[0] + gl_arrs[i+8]*arrp[1] +
                 gl_arrs[i+6]*arrp[2] + gl_arrs[i+3]*arrp[3];

    for (j = 0; j<4; ++j)
      arrq[j] += arrb[j][i];
   }
}
```

In the modified code, the calculation of `arrb[n][i]` is independent of the calculation of `arrb[n][i+1]`. This permits the compiler to distribute the `i` loop:

```
for (i = 0; i<62500; ++i)     /* VECTOR */
  {
   arrb[0][i] = gl_arrs[i  ]*arrp[0] + gl_arrs[i+4]*arrp[1] +
                gl_arrs[i+7]*arrp[2] + gl_arrs[i+9]*arrp[3];
   arrb[1][i] = gl_arrs[i+4]*arrp[0] + gl_arrs[i+1]*arrp[1] +
                gl_arrs[i+5]*arrp[2] + gl_arrs[i+8]*arrp[3];
   arrb[2][i] = gl_arrs[i+7]*arrp[0] + gl_arrs[i+5]*arrp[1] +
                gl_arrs[i+2]*arrp[2] + gl_arrs[i+6]*arrp[3];
   arrb[3][i] = gl_arrs[i+9]*arrp[0] + gl_arrs[i+8]*arrp[1] +
                gl_arrs[i+6]*arrp[2] + gl_arrs[i+3]*arrp[3];
  }


for (j = 0; j<4; ++j)           /* Interchanged -- SCALAR */
  {
   V0= arrq[j];                 /* Hoisted register load  */
   for (i = 0; i<62500; ++i)    /* Interchanged -- VECTOR */
     arrq[j] += arrb[j][i];     /* Sunken register store  */
   arrq[j]=V0;
  }
```

The compiler vectorizes both distributed parts of the `i` loop. The second distributed part is interchanged with the `j` loop, which allows the compiler to hoist the load of `arrq[j]` and sink the corresponding store. These optimizations reduce the time required for each call to this routine by almost 90 percent.

Remove a Conditional from a Loop

A loop with an embedded conditional usually runs slower than a loop without a conditional, even if both loops are vectorized. In addition, some types of conditionals can prevent the compiler from vectorizing a loop. Remove conditional tests from loops whenever possible.

The compiler vectorizes both `i` loops in the following example. The second loop, however, has a series of imbedded `if` tests that slow it down.

```
#include <fastmath.h>

iftest()
{
 float a1[10000], b1[10000], c1[10000];
 int i;

 for (i=0; i<10000; i++)
   {
    a1[i]= 1.0;
    c1[i]= 0.0;
   }

 for (i=0; i<10000; i++)
   {
    if (i<2000)
      {
       c1[i]= a1[i]* 2000.0 + cos(a1[i]);
       b1[i]= b1[i]* c1[i]*c1[i]*c1[i]*c1[i]/ a1[i];
      }
    if ((i>2000) && (i<4000))
      {
       c1[i]= a1[i]+ cos(a1[i]);
       b1[i]= b1[i]+ c1[i];
      }
    if ((i>4000) && (i<6000))
      {
       c1[i]= a1[i]+ 2000.0;
       b1[i]= b1[i]* b1[i]* b1[i]* b1[i];
      }
    if (i>6000)
      {
       c1[i]= a1[i];
       b1[i]= 1.0;
      }
   }
}
```

Remove the conditional by distributing the second loop. This produces five distributed parts:

```
for (i=0; i<10000; i++)
  {
   a1[i]= 1.0;
   c1[i]= 0.0;
  }

for (i=0; i<2000; i++)
  {
   c1[i]= a1[i]* 2000.0 + cos(a1[i]);
   b1[i]= b1[i]* c1[i]*c1[i]*c1[i]/ a1[i];
  }

for (i=2000; i<4000; i++)
  {
   c1[i]= a1[i]+ cos(a1[i]);
   b1[i]= b1[i]+ c1[i];
  }

for (i=4000; i<6000; i++)
  {
   c1[i]= a1[i]+ 2000.0;
   b1[i]= b1[i]* b1[i]* b1[i]* b1[i];
  }

for (i=6000; i<10000; i++)
  {
   c1[i]= a1[i];
   b1[i]= 1.0;
  }
```

Each distributed part is vectorized by the compiler. The resulting code runs approximately 10 times faster than the original loop.

Tuning a Parallel Loop

This example is a modification of a function taken from a standard benchmarking suite and is compiled with `-alias array_args`. The subroutine calculates the product of vector `arrx` and matrix `arrm`.

```
prodxm(arry, n1, arrx, n2, arrm)
int n1, n2;
double arry[], arrx[], arrm[300][300];
{
 int i, j;

 for (j= 0; j<n2; ++j)
   for (i = 0; i<n1; ++i)
     arry[i] += arrx[j] * arrm[i][j];
}
```

The first `n1` elements of `arry` are assigned values according to the following equation:

```
arry[i] = arrx[1]*arrm[i][1] + arrx[2]*arrm[i][2] +
          arrx[3]*arrm[i][3] + ... + arrx[n2]*m[i][n2]
```

The compiler vectorizes the `i` loop, producing a loop nest with the `j` loop on the outside and a vector strip-mine loop inside:

```
for (j= 0; j<n2; ++j)
  for (i_outer=1; i_outer<n1; i+=128)
     arry[i_outer:min(i_outer+127,n1)] =
       arry[i_outer:min(i_outer+127,n1)] +
       arrx[j] * arrm[i_outer:min(i_outer+127,n1)][j];
```

Each iteration of the inner loop requires three vector operations: a register load of `arrm`, register load of `arry`, and register store to `arry`. These three chained vector operations execute in three chimes (*chained vector times*).

By interchanging the strip-mine loop with the `j` loop, the compiler is able to hoist a vector load above the `j` loop and sink a vector store below:

```
for  (i_outer=1; i_outer<n1; i_outer+=128)
  {
   V0= arry[i_outer:min(i_outer+127,n1)];
   for (j= 0; j<n2; ++j)
     V0 = V0 + arrx[j] * arrm[i_outer:min(i_outer+127,n1)][j];
  }
   arry[i_outer:min(i_outer+127,n1)] = V0;
```

Within the `j` loop, vector register `V0` serves as an accumulator. Because `arry` is constant within the `j` loop, the only memory access required is a vector register load of `arrm`. This reduces the time required for an iteration of the `j` loop to only one chime—a significant improvement in performance.

At optimization level `-O3`, the compiler generates code that spreads the work among the expected number of processors. In the following code, EP represents the number of processors found on the machine where the program is compiled or the number of processors specified with the `-ep` option). `sl` is the maximum strip length for parallel execution.

```
sl = max(n1/(2*EP),1);
for (i_outer=1; i_outer<n1; i_outer+=sl)          /* PARALLEL */
  {
   V0= arry[i_outer:min(i_outer+(sl-1),n1)];
   for (j= 0; j<n2; ++j)
     V0 = V0 + arrx[j] * arrm[i_outer:min(i_outer+(sl-1),n1)][j];
  }
   arry[i_outer:min(i_outer+127,n1)] = V0;
```

If `n2` is small and `n1` is large, this code is highly efficient. If `n1` is small and `n2` is large, it is better to vectorize the `j` loop. If `n1` is greater than 256 but smaller than 384, the outer loop divides the work into three parallel strips; this is inefficient on a machine with two or four CPUs. If `n1` is less than or equal to 128, only one strip is created, and the possibility of parallelism, with its benefit of reduced turnaround time, is lost.

Now look at the calling function:

```
call_prodxm()
{
 int i;
 arry[300], arrx[300], arrm[300][300];

 for (i= 0; i<300; ++i)
   prodxm(arry, i, arrx, 299-i, arrm);
}
```

Notice that the prodxm function is called 300 times. On the first call, the value of `n1` is 0 and the value of `n2` is 300. `n1` is incremented and `n2` decremented prior to each succeeding call.

Further examination of the code shows that this function has the most work to do when `n1` and `n2` are approximately equal (when the value of both variables is close to 150). When `n1` is 150, the code creates parallel strips of length 22 and 128. Parallelism is most efficient, however, when strip lengths are nearly equal, giving the processors equal amounts of work to do. Setting the maximum strip length to 75 results in strips of equal length when `n1` and `n2` are equal. This results in a less favorable strip length when `n1` is large, but has little effect when `n1` is small. On the whole, it seems like a reasonable optimization to try.

Use the `vstrip` directive to force a maximum strip length of 75 on the `i` loop:

```
prodxm(arry, n1, arrx, n2, arrm)
int n1, n2;
double arry[], arrx[], arrm[300][300];
{
 int i, j;

 for (j= 0; j<n2; ++j)
   /* $dir vstrip(75) */
   for (i= 0; i<n1; ++i)
     arry[i]= arry[i]+ arrx[j]* arrm[i][j];
}
```

This produces a small improvement in performance. More improvement is possible by vectorizing the `j` loop instead of the `i` loop when `n1` is small. Using the `scalar` and `force_vector` directives, as shown below, you can create a vectorized `j` loop that is executed only when the value of `n1` is small:

```
prodxm(arry, n1, arrx, n2, arrm)
int n1, n2;
double arry[], arrx[], arrm[300][300];
{
 int i, j;

 if (n1 > 11)
   {
    for (j= 0; j<n2; ++j)

    /* $dir vstrip(75) */
    for (i = 0; i<n1; ++i)
      arry[i] = arry[i] + arrx[j] * arrm[i][j];
   }
 else
   {
    /* $dir force_vector */
    for (j= 0; j<n2; ++j)

    /* $dir scalar */
    for (i = 0; i<n1; ++i)
      arry[i] = arry[i] + arrx[j] * arrm[i][j];
   }
}
```

The cutoff point for executing the vectorized loop is determined by experimenting with various values of `n1` and timing the results using CXpa.

Many loops that are automatically parallelized are the strip-mine loops that result from vectorization. Strip-mine loops, however, only present a good balance of work when the actual vector length is great. For small trip counts that typically fall within a narrow range and for constant trip counts, use the `VSTRIP` directive to shorten the strip-mine loop. For small trip counts that vary widely and for unknown trip counts, use the `-ds` option to turn on automatic dynamic selection.

Strip-Mine a Loop by Hand

Sometimes, the only way to vectorize a loop is to strip-mine the loop by hand. The following function finds the maximum absolute value from all the elements in an array pointed to by a.

```
#include <fastmath.h>

float vmaxabs(a,n)
float *a;
int   n;
{
 int i;
 float max;
 int imax;

 for (imax=0, i=1; i<n; i++)
   if ( sfabs(a[i])>sfabs(a[imax]) )
     imax=i;

 max= sfabs(a[i]);
 return(max);
}
```

The scalar variable imax causes an LCD that prevents the compiler from vectorizing this loop. The only way to vectorize the loop is to rewrite it to process the array as strips of 128:

```
#include <fastmath.h>

float vmaxabs(a,n)
float *a;
int   n;
{
 static float arrb[128];
 float max = (float) 0.0;
 int i,j,k,m,sl;

 for (i=0; i<n; i+=128)
   {
    sl = ( i+128<n )? 128: n-i;

    /*$dir no_recurrence, max_trips(128) */
    for (m=0, j=i; m<sl; m++,j++)
      arrb[m] = sfabs(a[j]);

     /*$dir no_recurrence, max_trips(128) */
    for (j=0, k=1; k<sl; k++)
      if (arrb[k]>arrb[j])
        j=k;

    if (arrb[j]>max )
      max = arrb[j];
   }
 return(max);
}
```

In the modified code, the `i` loop iterates by 128. On each iteration but the last, the strip length (`sl`) is set to 128. On the last iteration, the strip length is set to 128 or to the number of elements remaining to be processed in the array (`n-1`), whichever is less.

The `m` loop calculates the absolute values of the elements in the current strip. Because the loop is strip-mined by hand, a `max_trips` directive is added. Setting the value of `max_trips` to 128 prevents the compiler from generating strip-mine overhead for this loop.

The `j` loop finds the maximum value from the `arrb`, where the `m` loop stores the absolute values. This `j` loop has a pattern that is matched by the compiler. Pattern-matching is a potentially unsafe optimization, so this loop is vectorized only if compiled with the `-uo` option.

The maximum value in `arrb` is then compared to `max`, which holds the maximum value found in preceding strips.

Note that the constant 0.0 is cast to type `float` before it is assigned to `max`. Floating-point constants are normally stored as type `double`. This cast avoids a runtime type conversion.

Substitute Lookup for Computation

One of the simplest ways to increase the speed of a routine is to replace computations with table lookup of precomputed values. The following example counts the leading zeros in a bit field:

```
int count_leading(a)
int a;
{
 int i;

 a <<= 8;
 for ( i = 0; i < 4; i++ )
   {
    if ( a & 0x80000000 )
      return(i);
      a <<= 1;
   }
    return(4);
}
```

This function can be performed much more efficiently using table lookup:

```
int count_leading_2(a)
int a;
{
 static int shift_table[16] =
   {
    4,   /* bit pattern 0000   */
    3,   /* bit pattern 0001   */
    2,   /* bit pattern 0010   */
    2,   /* bit pattern 0011   */
    1,   /* bit pattern 0100   */
    1,   /* bit pattern 0101   */
    1,   /* bit pattern 0110   */
    1,   /* bit pattern 0111   */
    0,   /* bit pattern 1000   */
    0,   /* bit pattern 1001   */
    0,   /* bit pattern 1010   */
    0,   /* bit pattern 1011   */
    0,   /* bit pattern 1100   */
    0,   /* bit pattern 1101   */
    0,   /* bit pattern 1110   */
    0    /* bit pattern 1111   */
   };

 return(shift_table[(a >> 20) & 0x0f]);
}
```

Conclusion

Optimization remains more of an art than a science. Optimization tricks presented in this chapter can be applied to optimizing your own program. When you understand the principles involved and begin using them, you will undoubtedly discover optimization tricks of your own.

Chapter 9
Limitations of Optimization

Optimization changes the order in which instructions execute. The goal is to produce the fastest possible code without sacrificing accuracy. In some cases, however, optimization can produce the following effects:

- Wrong answers.
- Code that runs slower than it did at lower optimization levels.

If you encounter either of these problems, use this chapter as a troubleshooting guide.

Wrong Answers

When a program produces different answers at different optimization levels, look for the following possible causes:

- Hidden aliasing.
- Floating-point imprecision (roundoff error).
- Misused directives and options.
- Compiler limitations.
- Nondeterminism of parallel execution.
- Numeric instabilities in your algorithm.

Hidden Aliasing

A hidden alias is the most common cause of answers that change from one optimization level to another. In C, aliases most often occur through the use of pointers. Aliasing is possible any time you use a pointer (*), address (&), or structure pointer (->) operator. Aliasing can also occur when you use array parameters, which, for efficiency, are always passed as pointers in C. Aliasing usually prevents the compiler from vectorizing a loop. If you use the `-alias array_args` option, aliasing of array parameters is hidden. This can cause the compiler to vectorize or parallelize a loop when it is not safe to do so.

The following program contains an alias. The formal parameters `arr1` and `arr2` are assigned the same actual parameter, `arrk`, at runtime. This causes a recurrence in the `i` loop, which should prevent vectorization.

```
confused(arr1, arr2)
int arr1[], arr2[];
{
int i, j;

for (i=1; i<128; i++)
  arr1[i]= arr1[i]+ arr2[i-1];

for (j=0; j<100; j+=10)
  {
   printf("arr1[%2d]= %d ", i, arr1[i]);
   printf("arr2[%2d]= %ad\n", i, arr2[i]);
  };
}

main()
{ int i, arrk[128];

  for (i= 0; i<128; i++)
    arrk[i]= 1;

  confused(arrk, arrk);
}
```

When this program is compiled at `-O1` and executed, you see the following results:

```
% cc -O1 alias.c
% a.out
arr1[ 0]=  1    arr2[ 0]=  1
arr1[10]= 11    arr2[10]= 11
arr1[20]= 21    arr2[20]= 21
arr1[30]= 31    arr2[30]= 31
arr1[40]= 41    arr2[40]= 41
arr1[50]= 51    arr2[50]= 51
arr1[60]= 61    arr2[60]= 61
arr1[70]= 71    arr2[70]= 71
arr1[80]= 81    arr2[80]= 81
arr1[90]= 91    arr2[90]= 91
```

If you compile the program at -O2 without the -alias array_args option, no vectorization occurs and you get the same results. If you compile the program at -O2 with the -alias array_args option, however, you get the following output:

```
% cc -O2 -alias array_args -or none alias.c
% a.out
arr1[ 0]=  1    arr2[ 0]=  1
arr1[10]=  2    arr2[10]=  2
arr1[20]=  2    arr2[20]=  2
arr1[30]=  2    arr2[30]=  2
arr1[40]=  2    arr2[40]=  2
arr1[50]=  2    arr2[50]=  2
arr1[60]=  2    arr2[60]=  2
arr1[70]=  2    arr2[70]=  2
arr1[80]=  2    arr2[80]=  2
arr1[90]=  2    arr2[90]=  2
```

The function confused expects to receive two arrays as parameters, but the main function has passed it the same array twice. This produces a dependency in the i loop: the calculation of the nth element depends on the previous calculation of the (n-1)th element. The -alias array_args directive assures the compiler that the two arrays are distinct. The compiler ignores the possible dependency and vectorizes the loop. The vectorized loop adds the (n-1)th element of the loop to the nth element, just as the scalar version did, but does all of the adds simultaneously, using the original value of the (n-1)th element instead of the calculated value. As a result, the calculated values for each nth element are changed.

Illegal Subscripts

The value of a subscript expression must be greater than or equal to the lower bound of the array, which is zero in C, and less than or equal to the upper bound, which is the size of the array minus one.

Subscripts that go out of bounds are a frequent cause of answers that vary between optimization levels and programs that abort and dump core.

Iterating by Zero

If the compiler vectorizes a loop that iterates a variable by zero on each trip, that loop can produce incorrect answers or cause the program to abort and dump core. This can happen when a variable is used as an iteration value. The following example shows three loops that iterate by zero:

```
for (i=0, j=0; i<n; i++, j+= zed)
  {
   arrb[i]= arra[j];
   arra[j]= arrc[i];
  }

for (i=0; i<n; i+= zed)
  arra[i]=arrb[i];

for (i=0; i<n; i++)
  {
   arrb[i]= arra[j];
   j+= zed;
   arra[j]= arrc[i];
  }
```

In this example, `zed` is an integer variable that takes the value of zero, causing the loops to fail. All three loops vectorize when compiled at `-O2`. The first loop runs, even when vectorized, but produces different results than it did at `-O1`. The other two loops, which are infinite when compiled at `-O1`, cause the program to abort and dump core when compiled at `-O2`.

If the compiler can tell that a loop iterates by zero, it does not vectorize the loop. This problem occurs only when the compiler cannot predict the iteration value. This problem usually occurs when the iteration value is determined by a global variable or a parameter passed in from another function.

Floating-Point Imprecision, or Round-Off Error

The compiler applies normal arithmetic rules to real numbers. Arithmetically equivalent expressions are assumed to produce the same results.

Most floating-point numbers cannot be represented in hardware with 100 percent accuracy, but are rounded off to the nearest value that can be represented. When optimization changes the evaluation order of a floating-point expression, it can also change the results. Possible consequences of floating-point round-off include program aborts, divide-by-zero errors, address errors, and answers that vary by a large percentage.

Floating-point precision problems frequently occur when a program tests the value of a variable without allowing sufficient tolerance for round-off errors. This problem can usually be solved by allowing greater tolerance in the test or by making the variable double-precision (`double`) instead of single-precision (`float`).

At optimization level `-O2`, round-off problems are caused by differences between the rounding algorithms used in the vector and scalar processing units. An expression evaluated to 32-bit precision in scalar may yield a positive or negative number very close to zero, while the same expression evaluated to 32 bits in vector unit yields zero. When evaluated to 64-bit precision, the answers may agree more closely, but 32-bit and 64-bit answers usually differ significantly.

Reductions can present a problem because they change the order in which an operator is applied to the values in the vector. This reordering improves performance, but can change results. This problem, too, can frequently be solved by using greater precision.

The compiler can reorder code in an expression to increase the efficiency of evaluation. Reordering can cause problems if your program assumes that associative operations are evaluated in a specific order. Consider the following example:

```
float x1, x2, a, b, c;

a= 1.33e-32;
b= 1.55e-32;
c= 2.22e+36;

x1= a* b* c;                          /* Expression 1 */
printf(" x = %12.8e\n", x1);

x2= c* b* a;                          /* Expression 2 */
printf(" x = %12.8e\n", x2);
```

Expressions 1 and 2 are mathematically equivalent. When evaluated with infinite precision, they produce the same result. Because data types in C (and most other programming languages) have finite precision, two mathematically equivalent expressions may not always produce the same result.

In this particular example, multiplying `a` by `b` produces an underflow. Because the actual value is too small to be stored as a 32-bit floating-point number, the computer represents the result as zero. Multiplying zero by `c` also produces result that is zero. If `b` and `c` are multiplied first, no underflow results.

If you assume standard left-to-right evaluation for expressions 1 and 2, you should expect the program to produce the following output:

```
x1 = 0.00000000e+00
x2 = 4.57653003e-28
```

Reordering of the code, however, may invalidate this assumption. In this example, Expression 2 is evaluated left to right, but Expression 1 is evaluated right to left. This reordering occurs even at optimization level `-no`. Because of reordering, the program actually produces the following results:

```
x1 = 4.57653003e-28
x2 = 4.57653003e-28
```

Sometimes you may want the result of Expression 1 to be zero. In ANSI C mode, you can use parentheses to force the order of evaluation:

```
float x1, x2, a, b, c;

a= 1.33e-32;
b= 1.55e-32;
c= 2.22e+36;

x1= (a*b)* c;                         /* Expression 1 */
printf(" x = %12.8e\n", x1);

x2= c* b* a;                          /* Expression 2 */
printf(" x = %12.8e\n", x2);
```

This ANSI C program yields a value of zero for Expression 1. When the program is compiled in the backward-compatible mode, however, the result of Expression 1 is still 4.57653003e-28. This is because the non-ANSI C (Kernighan and Ritchie) rules permit the compiler to ignore parentheses when determining the order of evaluation. Another way to force a specific evaluation order is to break the expression up into multiple statements:

```
float x1, x2, a, b, c;
float temp;

a= 1.33e-32;
b= 1.55e-32;
c= 2.22e+36;

temp= a * b;                          /* Expression 1a */
x1= temp* c;                          /* Expression 1b */
printf(" x = %12.8e\n", x1);

x2= c* b* a;                          /* Expression 2 */
printf(" x = %12.8e\n", x2);
```

Note

The `-parens` option can effect the way parentheses are treated in both ANSI and non-ANSI modes. If you specify `-parens explicit`, parentheses are honored regardless of the compilation mode. If you specify `-parens ignore`, parentheses are ignored and the compiler can reorder any integer or floating-point expression. This is the default in `-pcc` and `-ext` modes. If you specify `-parens implicit`, the compiler honors all parentheses, as well as grammer and associativity rules, for floating-point expressions and no reordering can be performed. This is the default for all other compilation modes.

These options affects only floating-point expressions. Integer expressions can always be reordered in any compilation mode.

Misused Directives and Options

Misused directives are a common cause of incorrect answers. Parallelizing a loop that contains a function call, for example, is safe only when that function contains no dependencies that might cause a recurrence.

Do not assume that it is safe to parallelize any loop that can be safely vectorized. A loop can be safely vectorized as long as it does not contain a backward LCD. Forward or backward LCDs, as well as LIDs, can prevent a loop from being safely parallelized.

The main function in the following example initializes `arra`, calls calc, and displays the new array values. In the function calc, there is a potential recurrence on `(a[i+n])` that prevents the compiler from vectorizing the `i` loop.

```
float arra[1025], arrb[1025];
main()
{
 int i;

 for (i=0; i<1025; ++i)
   arra[i]= i;
 calc(1);
 for (i=0; i<1025; ++i)
 printf("%d  %d  \n", i, arra[i]);
}

calc(n)
{
 int i;

 for (i=0; i<1024; ++i)
   arra[i] = arra[i + n] + arrb[i];
}
```

Because you know that the value of `n` is always 1, you can use the `no_recurrence` directive to tell the compiler that the loop in calc can be vectorized safely:

```
calc(n)
{
 int i;

 /* $dir no_recurrence */
 for (i=0; i<1024; ++i)
   arra[i] = arra[i + n] + arrb[i];
}
```

Safe vectorization does not imply safe parallelization. Using the `force_parallel` directive on the same loop is inappropriate.

```
calc(n)
{
 int i;

 /* $dir force_parallel */         /*** <-- MISUSED DIRECTIVE ***/
 for (i=0; i<1024; ++i)
   arra[i] = arra[i + n] + arrb[i];
}
```

The compiler warns you of the dependency but parallelizes the loop. The parallel code may produce incorrect results.

Compiler Limitations

Reductions, discussed more fully in Chapter 3, are a special class of recurrences that can be vectorized. An apparent recurrence may prevent the compiler from vectorizing a loop containing a reduction. The following loop is not vectorized because of the apparent dependency between the reference to `arra[i]` and the assignment to `arra[arrx[j]]` in the next line.

```
for (i=0; i<5; ++i)
  for (j=0; j<5; ++j)
    {
     arra[i]+= arrb[j] * arrc[j];
     arra[arrx[j]] = arrb[j] + arrc[j];
    }
```

A `no_recurrence` directive placed before the `j` loop tells the compiler that the indirect subscript does not cause a true recurrence. It also tells the compiler to ignore the reduction on `arra[i]`. The compiler generates normal vector load, add, and store instructions for the first statement. The resulting code runs fast and produces incorrect answers.

To solve this problem, distribute the `j` loop, isolating the reduction from the other statements:

```
for (i=0; i<5; ++i)
  for (j=0; j<5; ++j)
    arra[i] = arra[i] + arrb[j] * arrc[j];

  for (j=0; j<5; ++j)
    arra[arrx[j]] = arrb[j] + arrc[j];
```

The apparent recurrence is removed and both loops vectorize. This problem occurs only if the reduction and the apparent recurrence involve the same variable. If they involve different variables, as in the following example, both the reduction and the recurrence are handled correctly:

```
for (i=0; i<5; ++i)
  /* $dir no_recurrence */
  for (j=0; j<5; ++j)
    arra[i] = arra[i] + arrb[j] * arrc[j];

  for (j=0; j<5; ++j)
    arra[arrx[j]] = arrb[j] + arrc[j];
```

Nondeterminism of Parallel Execution

In a parallel program, threads do not execute in a predictable order. If you force the compiler to parallelize a loop when a dependency exists, the results cannot be predicted and may vary from one execution to the next. The errors are nondeterministic and depend on the order in which statements are executed. Unless you are sure that a recurrence does not exist, it is safer to let the compiler choose which loops to parallelize.

Slower Code

When your program takes longer to execute at a higher level of optimization than it did at the previous level, look for:

- Misused compiler directives.
- Short vector length (small trip count).
- Complicated conditionals in a loop nest.
- Parallel overhead when a program compiled at `-O3` runs on a single CPU.

Misused Directives

The `synch_parallel` directive tells the compiler to parallelize a loop and insert synchronization code to ensure that dependencies are honored. Not all parallel loops can be synchronized efficiently. Usually, the compiler generates more efficient code automatically than any code you can create with the `synch_parallel` directive.

For synchronized code to be efficient, the independent (parallel) part of a loop must be larger than the dependent (sequential) part.

The compiler calculates the optimum vector and strip lengths based on the number of CPUs detected or specified with the `-ep` option. When you overrule the compiler by using the `vstrip` and `pstrip` directive, be sure the strip length you have chosen is really more efficient.

Short Vector Length

The compiler automatically vectorizes a loop if it has more than two iterations or if the loop trip count is unknown. A loop with only three or four iterations may run slower when it is vectorized. The `scalar` directive can be used to prevent the compiler from vectorizing short loops. The `select` directive tells the compiler to generate multiple versions of a loop and code that allows dynamic (runtime) selection of the best version. Using the `select` directive, you can specify cutoff points for scalar, vector, and parallel processing.

Complex Conditionals

Nested loops that contain elaborate conditionals may run slower at optimization level `-O2` than at `-O1`.

When the compiler vectorizes a loop that contains an `if-else` statement, it creates a separate vector loop for each clause. Instead of choosing one of these loops to execute at runtime, the program executes both. Results from these two loops are merged using a conditional mask to produce the final result. If there is not a good balance between the amount of code in each alternative, the overhead can be very great. This is especially true when the clause containing the smaller amount of code is branched to more often.

A short vector length (small trip count) makes a loop of this type even less efficient. Simplify conditionals, remove them from the loop, or use the `scalar` directive to prevent vectorization.

Conclusion

Optimization changes the order in which instructions execute. Compiler directives change the order still further.

These changes sometimes cause your program to produce different answers or to slow down. If this happens, you may need to recompile part of your program at a lower optimization level, redeclare variables, modify your code, or remove certain directives.

APPENDIX A
The -uo Option

The -uo option on the cc command line instructs the compiler to attempt potentially unsafe optimizations. The following optimizations are enabled by the -uo option:

- Simple strength reductions.
- Code motion.
- Pattern matching.
- Elimination of type conversion.

Simple Strength Reduction

Chapter 2 describes how the compiler replaces certain slow operations with faster ones on the assumption that mathematically equivalent expressions produce equivalent results.

Reducing an expression such as `x / C` to `(1 / C) * x`, may be unsafe because it can increase round-off errors.

When the -uo option is used, the compiler replaces division operations with multiplication, unless a possibility of overflow exists.

Code Motion

The compiler normally moves an invariant expression out of a loop if the expression is located on a path to all loop exits. When -uo is used, the compiler can move an invariant expression out of a loop if the expression is located on a path to any one loop exit.

Pattern Matching

Pattern matching allows the compiler to vectorize certain loops that could not otherwise be vectorized. The compiler recognizes the following loops as having patterns on which it can perform reductions:

```
for (i=0; i<n; i++)
  arrx[i] = arrx[i-1]+ arry[j];

for (imax=0, i=1; i<n; i++)
  if (arrx[i] > arrx[imax])
    imax= i;

for (imin=0, i=1; i<n; i++)
  if (arrx[i] < arrx[imin])
    imin= i;

for (a_index= -1. i=0; i<n; i++)
  if (arrx[i] == y)
    a_index= i;
```

Conversion Elimination

When the -uo option is used, the compiler eliminates costly type conversions by creating real induction variables, as shown in the following example:

Original Loop	Optimized Loop
`for (i=1; i<100000; ++i)` `  arra[i] = i;`	`for (real_i = 1.0; real_i<100000; ++real_i)` `arra[real_i] = real_i;`

This potentially unsafe optimization affects only scalar loops. At optimization level -O2, the example loop shown above is vectorized normally.

APPENDIX B
Glossary

aliases

Aliases are alternative names for the same object—in the context of this document, a variable or memory location. Aliases in C usually arise through the use of pointers or array parameters.

ASAP

Automatic Self-Allocating Processors, a proprietary feature found only on CONVEX hardware that allows CPUs to seek out and execute the next piece of work with a minimum of overhead.

bank conflict

An attempt to load two elements from the same memory bank. On CONVEX C200 Series machines, each memory board is divided into four 64-bit memory banks. Each bank holds one double-precision or two single-precision numbers. Arrays are stored in main memory across all four banks.

Loading each array element takes four clock cycles, during which time no other element can be retrieved from the same bank. Storing contiguous array elements across four memory banks allows each of the three intervening clock cycles to be used for loading another element. If the loading of an element from bank 0 begins on cycle `c`, the next element (in bank 1) can start loading on cycle `c+1`, the next (from bank 2) on cycle `c+2`, and the next (from bank 3) on cycle `c+3`. By cycle `c+4`, the first load has completed and another load from bank 0 can begin.

basic block

A basic block is a linear sequence of statements that ends with a conditional or unconditional branch. There can be no branches within the body of a basic block. A C function contains at least one basic block and typically contains many.

chaining

(See *vector chaining.*)

chime

A chained vector time. The time required to perform the simultaneous instructions of one vector chain. On the CONVEX C100 and C200 Series machines, this is equal to the vector length plus 10 clock cycles.

communication register

A high-speed register used for communication among threads of a process. Threads communicate by sending and receiving data through the communication registers. A lock bit maintained by hardware is associated with each communication register; the lock bit guarantees mutually exclusive access to the register.

concurrent

In parallel processing, threads that may execute at the same time are concurrent.

CPU

A central processing unit. The CONVEX C200 Series provides up to four CPUs on each machine.

critical region

A segment of code that modifies data or computer resources shared by threads. Only one CPU or process should be executing a given critical region at the same time. If this restriction is not maintained, the program containing the critical region produces nondeterministic results (usually errors).

dependency

A dependency is a relationship between two operations such that one operation depends on the results of the other. This implies a definite time-ordering of the operations: the execution of one operation must always precede the execution of the other, and they cannot be interchanged. (See also *loop-carried dependency* and *loop-independent dependency.*)

execution stream

A series of instructions executed by a CPU.

functional unit

A unit within a CPU that carries out a specific set of instructions. Multiple functional units allow the processor to execute multiple instructions concurrently. The C200 Series processor has three functional units, each of which performs a separate set of instructions:

* Add/subtract, compare, shift, bit count, type conversion, and logical operations.
* Multiply/divide, square root, edit.
* Load/store.

granularity

Parallelism can be classified according to the size and number of threads. A program with a few, very large threads is said to be *coarse-grained.* A program with many, very small threads is said to be *fine-grained.* This property is called *granularity.* Depending on the method of parallelization, the size of a thread can range from half a program (extremely coarse) to a single source statement (fine-grained).

hoisting

The lifting of a variable, array reference, or piece of code out of a loop to a basic block preceding the loop. (See also sinking.)

loop-carried dependency (LCD)

A loop-carried dependency (LCD) exists when one iteration of a loop assigns a value that is referenced during a later iteration. For example:

```
for (i=1; i<=n; i++)
  a[i+2] = a[i] + 3.14;
```

On the first iteration of this loop, the value of `a[1] + 3.14` is assigned to `a[3]`; on the third iteration, the value of `a[3] + 3.14` is assigned to `a[5]`. The third iteration uses the value assigned in the first; the computation of `a[5]` cannot take place until the value of `a[3]` has been computed and assigned.

loop constant, or loop invariant

A variable, constant, or expression whose value does not change within a loop.

loop distribution

Creation of two or more loops, at the same nesting level, from an existing single loop. Loop distribution can increase the number of inner loops that the compiler vectorizes. It can also enable the compiler to interchange loops.

loop-independent dependency (LID)

A loop-independent dependency (LID) exists when two statements must be executed in a specific order within a single iteration. The following loop contains two such LIDs:

```
for (i = 0; i<n; i++)
  {
   a[i] = b[i] + c[i];
   b[i] = 0;
   c[i]++;
  }
```

Here, proper evaluation of the first statement, which assigns to `a`, can occur only if the next two statements, which alter the values of `a` and `b`, have not yet been evaluated.

loop induction variable

A variable whose value is conditionally or unconditionally incremented or decremented with each iteration of a loop. Loop-control variables are usually induction variables, but not all induction variables are loop-control variables. For example, both `k` and `i` are induction variables in the following loop:

```
k = 0;
for (i = 0; i<100; i++)
  if (cond(i))
    {
     a[i] = b[k];
     c[k] = d[i];
     k++;
    }
```

loop interchange

Loop interchange changes the nesting order of loops: an outer loop is moved inside an (originally) inner loop. Loops are interchanged to produce a more efficient vector strip length, to avoid a dependency, to increase efficiency of array accesses, or to increase granularity of a parallel outer loop.

multiprocessor	On a CONVEX supercomputer, the memory system, the I/O system, peripherals, and two or more CPUs.
mutual exclusion	A protocol or requirement that prevents simultaneous access to a given resource by multiple threads.
parallelization	Modification of code to enable loops to run simultaneously on multiple CPUs.
process	A collection of one or more execution streams within a single logical address space; an executable program. A process is made up of one or more threads. (See thread.)
recurrence	A cycle of dependencies among operations within a loop. (See *dependency.*)
reentrancy	The property of a function that enables multiple copies of the function to execute simultaneously. Each copy of a reentrant function maintains a thread-private copy of local data and a thread-private stack.
sinking	The movement of a variable, array reference, or piece of code out of loop to a basic block following the loop. (See also sinking.)
strip length, parallel	The increment by which the induction variable of an inner loop is advanced on each iteration of a parallel outer loop.
strip length, vector	The number of array elements processed in a given vector operation.
strip mining	An optimization that partitions operations into strips with no more than 128 elements each. Operations on these strips are controlled by an outer *strip-mine loop* created by the compiler.
synchronization	The coordination of two or more threads of parallel execution to prevent simultaneous access to the same critical region or to honor timing relationships (dependencies) in the code. Synchronization may be accomplished by compiler directives or assembly-language instructions. Synchronization entails some overhead when one CPU must wait for another.
thread	An independent execution stream that a CPU fetches and executes. One or more threads, each of which can execute on a different CPU, make up each process. Memory, files, signals, and other process attributes are generally shared among threads in a given process, enabling the threads to cooperate in solving the common problem.
thread-private, or thread-specific	Data that is accessible by a single thread only (not shared among the threads constituting a process). Thread-specific data use the same virtual address to refer to different physical memory locations.
vector chaining	The overlapping of vector operations in the CPU. For example, in the case of a vector load followed by a vector add, the add may be started as soon as the first operands are available.
vector spill	The temporary storing of vector register contents to memory. A spill occurs when the number of vectors used in a calculation exceeds the number of vector registers available.

APPENDIX C
Reporting Problems

This appendix introduces the CONVEX Technical Assistance Center (TAC) and `contact` utility. The `contact` utility is an online system for reporting problems to the TAC. To learn `contact` by using it, enter **contact** at the system prompt and then answer the questions as they appear on the screen. To find out more about using `contact`, read through this appendix. It describes prerequisites and tips for using `contact` and the step-by-step process `contact` takes you through.

Technical Assistance Center

The CONVEX Technical Assistance Center (TAC) is staffed by technical specialists who can address diverse questions and problems that arise in a supercomputing environment. If you have a hardware, software, or documentation question, contact the TAC. This group stands ready to solve such problems.

The contact Utility

The TAC recommends using the `contact` utility to report a hardware, software, or documentation problem. The `contact` utility is an interactive program that helps the TAC track reports and route them to the CONVEX personnel most qualified to fix a problem.

After you invoke `contact`, it prompts you for information about the problem. When you finish your report, `contact` electronically mails it to the TAC. You are notified within 48 hours that the TAC has received your report.

Prerequisites

To use `contact` requires

- a UNIX-to-UNIX Communication Protocol (UUCP) connection to the TAC
- full path name of the program or utility in question
- version number of the program or utility in question

UUCP Connection

Before using `contact`, check with your system administrator to be sure there is a UUCP connection to the TAC. A UUCP connection allows files to be copied from one UNIX-based system to another. The `uucp` (UNIX-to-UNIX copy) command relies on either a dial-up or hard-wired UUCP communication line.

Finding the Program Path Name

To determine the full path name of the program or utility in question, use the `which` command. The next screen illustrates using the `which` command to find the full path name of the loader (`ld`) utility.

```
>which ld
/bin/ld
>
```

In this example, the full path name of the loader is /bin/ld.

For more information on the `which` command, refer to the which(1) man page. You can also use the `info` online information system. Enter **info which** at the system prompt.

If you use the C shell (`csh`), you can also use the `whence` command to find the program path name. The `whence` command works like `which`, but faster.

Finding the Program Version Number

To determine the version number of the program or utility in question, use the `vers` command. The next screen illustrates using the `vers` command to find the version number of the loader (`ld`) utility. Enter **vers**, then the path name of the program or utility.

```
>vers /bin/ld
/bin/ld: 7.0
>
```

In this example, the loader version number is 7.0.

For more information on the `vers` command, refer to the vers(1) man page. You can also use the `info` online information system. To do so, enter **info vers** at the system prompt.

Tips on Using the contact Utility

The `contact` utility is interactive and easy to use. This section lists tips to help use it efficiently. In particular, this section tells how to

- use a .contact file
- abort a contact session
- resubmit an aborted report
- suspend a contact session
- move from one prompt to another
- use tilde-escape sequences in the `contact` utility

Using a .contact File

When you invoke `contact`, it prompts for information regarding the problem. The first prompt is for your name, title, phone number, and company name. You can, however, create a .contact file to skip this first prompt. Follow these steps:

1. Create a .contact file in your home directory.

2. Enter your name, job title, phone number, and company name, each on a new line.

When you invoke `contact`, it automatically includes the .contact file as input for the first prompt and proceeds to the next prompt.

Aborting the Report

To abort a contact report, either press the interrupt key (usually CTRL-C) or choose the abort option when prompted by the `contact` utility. Using CTRL-C to abort does not save the contents of the report. Using the abort option saves the contents of the report in a file named dead.report in your home directory.

Submitting the dead.report File

After you abort a contact session, the `contact` utility saves the report in a file named dead.report in your home directory. Using the `contact` command with the `-r` option automatically merges the contents of the dead.report file into the new contact session. Enter

contact -r

and `contact` finds the dead.report file in your home directory and merges it into the contact report. You can then edit the report. When you end the editing session, `contact` returns to the final prompt, which asks you to review, edit, submit, or abort the report.

Suspending a Report

Sometimes it is necessary to stop in the middle of a contact report and return to the shell (for instance, to suspend the contact session to find the program path name or version number). To suspend the contact session, press CTRL-Z. To return to the contact session, enter **fg**. Using CTRL-Z and the `fg` (foreground) command lets you toggle back and forth between the `contact` utility and the shell. You cannot, however, use CTRL-Z and `fg` to toggle back and forth if you are using the Bourne shell (`sh`).

Ending a Response

The `contact` utility prompts for information pertinent to your hardware, software, or documentation question. Some prompts require one-line responses; to move to the next prompt, press RETURN. Other prompts require more than a one-line response; to move to the next prompt, press CTRL-D.

Tilde-Escape Sequences

The `contact` utility treats input beginning with a tilde (~) as a special sequence. The character following the tilde is considered a request for a special function. The following tilde sequences are recognized by `contact`:

~e	start the text editor (defined in the EDITOR environment variable)
~h	display a list of available tilde-escape sequences
~p	print the contact report to the terminal screen
~r *filename*	read the contents of *filename* as a response to the current prompt. Some prompts require only a one-line response. This tilde-escape sequence works only for prompts that allow more than a one-line response.
~~	insert a single tilde as the first character in the line

Using the contact Utility

The `contact` utility prompts for the following information:

- your name, title, phone number, and corporate name
- name and version of the product
- one-line summary of the problem
- detailed description of the problem
- priority of the problem
- instructions on how to reproduce the problem
- comments about the problem
- comments about the documentation related to the problem
- files to include in the contact report

The following is a step-by-step discussion of these prompts.

Step 1a To invoke the `contact` utility, enter **contact** at the system prompt. The system responds with a welcome message and a series of questions regarding your hardware, software or documentation question. The next screen illustrates the `contact` command and the system response.

```
>contact
Welcome to contact version 0.11 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
>
```

Step 1b If there is a .contact file in your home directory, **contact** skips the first prompt. The next screen illustrates the **contact** command and the system response when a .contact file is in your home directory.

```
>contact
Welcome to contact version 0.11 ()

Enter the name of the product involved
>
```

Step 2 The **contact** utility prompts for the version number of the product. If you do not know the version number, use CTRL-Z to suspend the session. Use the **which** (or **whence** if you use **csh**) and **vers** commands to find the version number of the product. Use the **fg** command to return to the session and enter the version number in the form X.X or X.X.X.X.

Step 3 The **contact** utility prompts for a one-line summary of the problem. This summary is the subject header in any further correspondence regarding the problem. Please make this summary as descriptive as possible in one line.

Step 4 The **contact** utility prompts for a detailed description of the problem. Please make this description as complete as possible. Include source code and a stack backtrace when possible. (Refer to the adb(1) or csd(1) man page for information on obtaining a stack backtrace.) The more information you provide, the quicker the TAC can isolate and solve the problem.

Step 5 The **contact** utility prompts for the priority of the problem. The next screen illustrates this prompt and priority levels from which to choose; you must enter a priority number.

```
Enter a problem priority, based on the following:
1) Critical    - work cannot proceed until the problem is resolved.
2) Serious     - work can proceed around the problem, with difficulty.
3) Necessary   - problem has to be fixed.
4) Annoying    - problem is bothersome.
5) Enhancement - requested enhancement.
6) Informative - for informational purposes only.
>
```

Step 6 The **contact** utility prompts for an explanation of how to reproduce the problem. Please include the command syntax and options you used and anything else you did to make the program run.

Step 7 The **contact** utility prompts for any other pertinent comments. Please include all relevant information.

Step 8 The `contact` utility prompts for suggestions regarding documentation supporting the product. Indicate whether documentation could be revised to address the problem.

Step 9 The `contact` utility asks for names of files necessary to reproduce the problem. The next screen illustrates the `contact` prompt and sample user response.

```
Are there any files that should be included in this report (yes | no)?
>yes
Please enter the names of the files, one to a line (^D to terminate)
>test.f
>~/subroutines/sub.f
>
```

Note

Tilde-escape sequences are not recognized in responses to this prompt. In `contact`, a tilde in this section means your home directory. This convention is based on use of the tilde for expanding file names in `csh`.

If files specified are small text files, they are automatically included in the contact report. If the files are too large to be included in this report, `contact` gives further instructions on how to submit these files.

To specify a directory, combine directory files into a single file using the `tar` command (refer to the tar(1) man page for further information) or enter each file name in the directory on a single line in the contact report.

Step 10 The `contact` utility prompts you to review, edit, submit, or abort the contact report. The next screen illustrates this prompt.

```
Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
>
```

Choose the number of the option you want to select. These options let you do the following:

Review — review the text of the contact report. You are then prompted again to select an option.

Edit — edit the text of the contact report. If you choose to edit the report, `contact` puts you in your default text editor.

Submit — sends the report to the CONVEX TAC. You are notified within 48 hours that the TAC has received the report. This option exits the `contact` utility and returns you to the shell.

Abort	saves the text of the report in a file named dead.report in your home directory. This option exits the `contact` utility and returns you to the shell.

Index

T

U

V

W

CONVEX C Optimization Guide
Document No. 720-001130-200, First Edition

Reader's Forum

Please use this form to submit comments or questions concerning the clarity and service of this manual. Constructive critical comments are most welcome and help us continue in our efforts to generate quality customer documentation. Please list the page number for questions or comments.

_____

_____

_____

_____

_____

_____

_____

_____

From:

Name_____ Title_____

Company_____ Date_____

Address and Phone No._____

FOR ADDITIONAL INFORMATION OR DOCUMENTATION:

Location	Phone Number
From all locations in continental U.S.	1(800)952-0379
From locations in Alaska, Hawaii, & Canada	1(214)497-4379
From all other locations	Contact nearest CONVEX office

Direct mail orders to:

CONVEX Computer Corporation
Customer Service
PO Box 833851
Richardson TX 75083-3851 USA

(Fold Here First)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CONVEX Computer Corporation
Customer Service
PO Box 833851
Richardson TX 75083-3851
USA

(Fold Here Second)

(Tape or Staple)