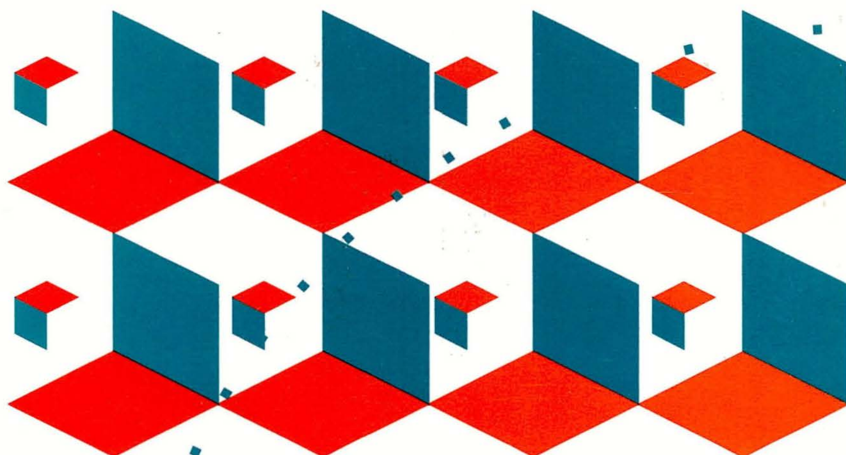


CONVEX

Application Compiler User's Guide

Third Edition

Convex Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214) 497-4000

Convex Application Compiler User's Guide

Order No. DSW-401

Third Edition

March 1995

Convex Press
Richardson, Texas
United States of America

Convex Application Compiler User's Guide

Order No. DSW-401

Copyright © 1995 Convex Computer Corporation
All rights reserved.

This document is copyrighted. This document may not, in whole or part, be copied, duplicated, reproduced, translated, electronically stored, or reduced to machine readable form without prior written consent from Convex Computer Corporation.

Although the material contained herein has been carefully reviewed, Convex Computer Corporation does not warrant it to be free of errors or omissions. Convex reserves the right to make corrections, updates, revisions or changes to the information contained herein. Convex does not warrant the material described herein to be free of patent infringement.

UNLESS PROVIDED OTHERWISE IN WRITING WITH CONVEX COMPUTER CORPORATION (CONVEX), THE PROGRAM DESCRIBED HEREIN IS PROVIDED AS IS WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES. THE ABOVE EXCLUSION MAY NOT BE APPLICABLE TO ALL PURCHASERS BECAUSE WARRANTY RIGHTS CAN VARY FROM STATE TO STATE. IN NO EVENT WILL CONVEX BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING ANY LOST PROFITS OR LOST SAVINGS, ARISING OUT OF THE USE OR INABILITY TO USE THIS PROGRAM. CONVEX WILL NOT BE LIABLE EVEN IF IT HAS BEEN NOTIFIED OF THE POSSIBILITY OF SUCH DAMAGE BY THE PURCHASER OR ANY THIRD PARTY.

Convex and the Convex logo ("C") are registered trademarks of Convex Computer Corporation.
Convex C100 Series and C200 Series are trademarks of Convex Computer Corporation.
ConvexOS is a trademark of Convex Computer Corporation.
Exemplar and Convex Exemplar are trademarks of Convex Computer Corporation.
SPP-UX is a trademark of Convex Computer Corporation.
C1, C120, C210, C220, C230, and C240 are trademarks of Convex Computer Corporation.
C3400, C3800, and C4600 are trademarks of Convex Computer Corporation.
UNIX is a registered trademark of Novell, Inc.
Other product names mentioned in this manual may be trademarks or registered trademarks of their respective companies, and are hereby acknowledged.

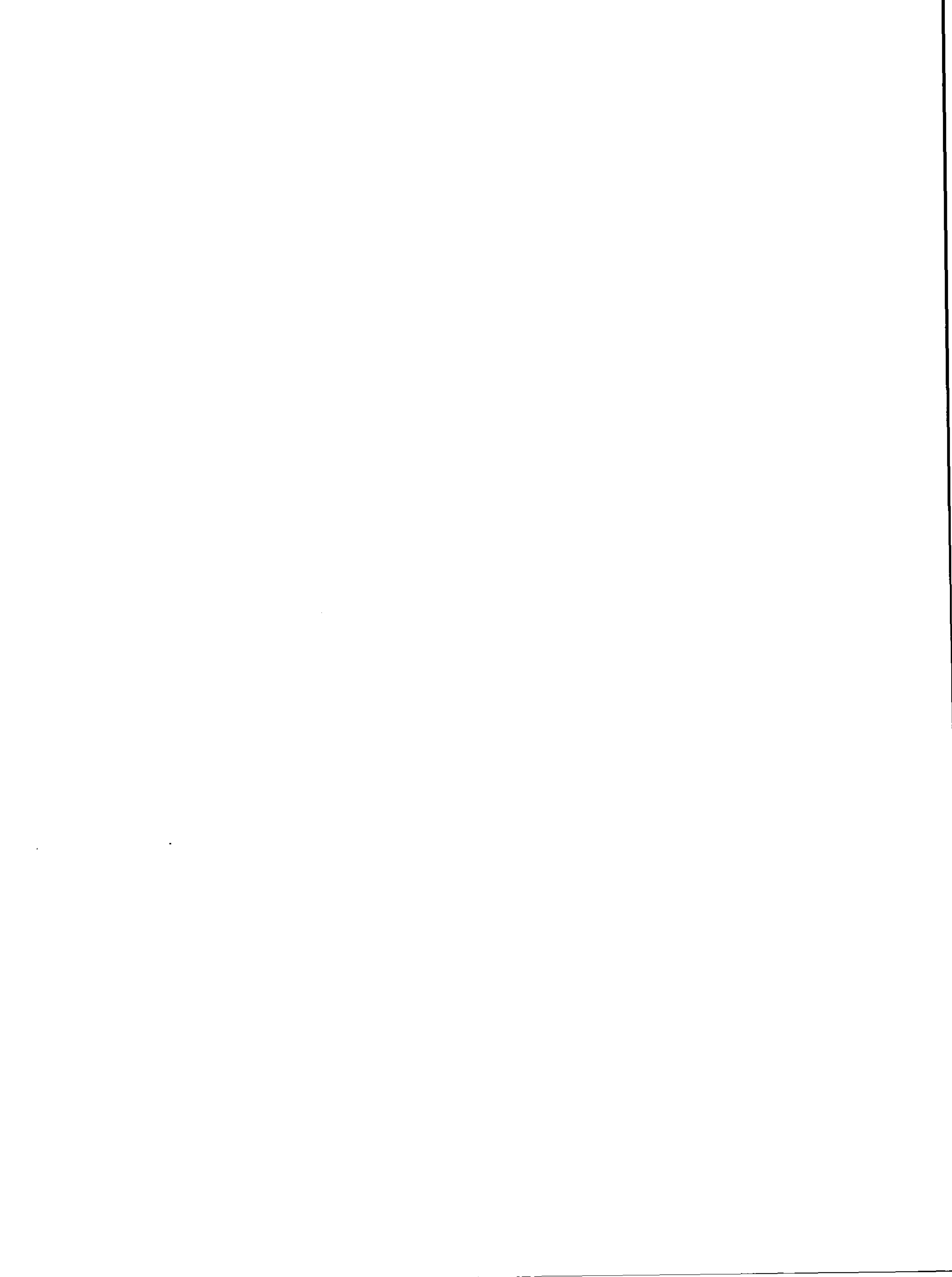
This entire book is recyclable.

Printed in the United States of America

Revision Information for

Convex Application Compiler User's Guide

Edition	Document No.	Description
Third	720-004030-004	Released March 1995 with Application Compiler version 2. Updated to include Exemplar-specific information and to incorporate the *PDBViewer Guide* as an appendix.
Second	720-004030-002	Released August 1992 with Application Compiler V 1.1.
First	720-004030-001	Released April 1991 with Application Compiler V 1.0.



Contents

How to use this book xv

Purpose and audience xv
Organization xv
Notational conventions xv
Associated documents xvi
Ordering documentation xvii
Technical assistance xviii

1 Principles and concepts 1

Basic optimization concepts 1
  -O0 optimizations 2
  -O1 optimizations 3
  -O2 optimizations 3
  -O3 optimizations 4
The Application Compiler 5
Application compiling 6
Problems of optimization 12

2 Basic interprocedural optimizations . 17

Interprocedural constant propagation 17
  Apparent recurrences 19
  Loop strides 20
  Loop interchange 21
  Dead code elimination 23
  Summary 24
Automatic inlining 25
  Improved data localization 25
  Improved loop parallelization 26
  Dead code elimination 27
Procedure cloning 28
Automatic parallelization of loops with calls 33
  Loops with calls parallelization vs.
  inlining and parallelizing 33
  Call eligibility 33
    Private data 34
    Self-dependence 35
    Procedure purity 38

Data privatization 39
Related procedural compiler directives 40
Manual data privatization 41
Forcing parallelization 43
Enhanced error checking 45
Alias and pointer tracking 46
Pointer tracking 48
Link optimization 51
Memory bank/cache line optimizations 51

3 Using the Application Compiler..... 53

Optimizing with the Application Compiler 53
Invoking apc 55
apc control options 55
Compilation options 56
Error control options 58
Message control options 64
Optimization options 84
Partial build options 88
Program database options 89
Profiling options 89
Optimization report 92
IPO report 96

4 Creating a buildfile 99

A simple buildfile 99
Buildfile statements 100
var_args statement 104
Specifying annotation mappings 105
The anno.map file 106
Libraries and linking 106
Linking indirectly called procedures 107
Directories 108
Flag macros and options statements 109
ANSI Fortran standard checking 112
Visual debugging 113
Combining C and Fortran 113

5 Creating libraries and object files . . 115

Building and using APC libraries 116
Using psum directives 117
Stub file format 117
A simple stub file 118
Compiling stub files 119

Psum directives 120
PSUM_ASGS 121
PSUM_KILLS 121
PSUM_USES 122
psum_range_names 122
psum_range_flags 123
psum_args_dealloc 123
PSUM_REENTRANT 124
PSUM_VAR_ARGS 124
PSUM_NO_IO 125
Example annotation 125
anno_ar utility 127
Examples 128
Creating and viewing an index 128
Extracting entry point information 129
Replacing/appending annotations 129

6 Directives 131
Inlining directives 131
INLINE 131
NO_INLINE 133
INLINE_CALL 134
NO_INLINE_CALL 136
Cloning directives 139
CLONE 139
NO_CLONE 144
Other directives 146
CONDITION_TRUE 146
ESTIMATED_TRIPS 147
FORCE_OBJECT 148
VAR_ARGS 149

7 Optimization strategy 151
Step 1: Scalar optimization 152
Step 1a: compile at -O1 152
Step 1b: check output 153
Step 1c: profile your program 154
Step 2: -O2 optimization 154
Step 2a: compile at -O2 154
Step 2b: check output 155
Step 2c: profile your program 155
Step 2d: look for further improvements 155
Step 2e: recompile 157
Step 3: -O3 optimization 158
Step 3a: compile at -O3 158
Step 3b: check output 158

Step 3c: profile your program 159
Step 4: Wrapping up 159

8 Fortran hints 161

Initialize COMMON-block variables 161
Watch for dynamic binding 163
Watch for misused Hollerith constants 166
Watch array bounds 166
Watch for undefined dummy variables 167
Don't use `loc()` pointer functions 168

9 C language hints 169

Take advantage of pointer tracking 169

Appendix A: Compiler options 171

Appendix B: The PDB Viewer interface 173

Graphical user interface 173
Invoking the PDB Viewer 173
Common menus 175
The File menu 175
The Scratchpad menu 175
Help menu 176
The main window 177
The File menu 177
The Reports menu 179
The Build menu 180
The Windows menu 181
Procedure windows 182
The Procedure List window 182
The Procedure Description window 183
Procedure source section 184
Messages section 185
Symbol Table section 187
Analysis and Symbol Information Display 187
Display control 188
The Call Information window 188
Editing the procedure source 190
The Symbol Table window 191
The Call Graph window 192
The File menu 193
The Windows menu 193
The Options menu 194

The View menu 196
The Messages window 198
X resources 199

Appendix C: Error and warning messages 201

Error messages 201
Interprocedural compilation errors 201
Buildfile errors 202
Variable problems 203
Warning messages 203
Argument problems 203
Definition problems 206
Pointer problems 207
Return-value problems 207
Procedure problems 208
Assignment problems 208
Other warnings 208

Glossary 211

Index 219

Figures

Figure 1 Control flow 6
Figure 2 Data flow 8
Figure 3 Summary phase 9
Figure 4 Synthesis/analysis/synthesis phases 10
Figure 5 Compile phase 11
Figure 6 Link phase 11
Figure 7 Loop without recurrence 13
Figure 8 Loop with recurrence 14
Figure 9 Loop without alias 47
Figure 10 Loop with alias 47
Figure 11 -vn option output 56
Figure 12 Time summary generated by -time option 58
Figure 13 IPO report with initialization errors 62
Figure 14 APC messages without -s 64
Figure 15 APC messages with -s option 65
Figure 16 Warnings generated by -show aliases 66
Figure 17 Array table generated by -show arrays 67
Figure 18 C-language array table 68
Figure 19 A call graph 69
Figure 20 Optimization report showing inlined procedures 72
Figure 21 Outline of procedure with loops 72
Figure 22 Outline of procedure with loops and calls 73
Figure 23 Pointer tables generated by -show pointers 74
Figure 24 Renamed variable messages 76
Figure 25 Renamed static variable message 76
Figure 26 Use and assign lists generated using -show scalars 77
Figure 27 Subroutine symbol table 78
Figure 28 Function symbol table 78
Figure 29 COMMON block symbol table 78
Figure 30 Messages generated with -check types and -show types 79
Figure 31 -show types messages generated when Fortran main calls C function 80
Figure 32 -show types messages generated when Fortran main calls DATE function 81
Figure 33 -show types messages generated when passing COMPLEX data type to C function 81

Figure 34 APC messages with -v option 83
Figure 35 APC optimization reports 93
Figure 36 Optimization report for
automatically inlined procedure 93
Figure 37 Optimization report for
procedure that can't be inlined 94
Figure 38 Optimization report for cloned procedure 95
Figure 39 IPO report 96
Figure 40 Example buildfile 99
Figure 41 Multiple main error message 109
Figure 42 Example source file organization 110
Figure 43 Buildfile for sources of Figure 42 111
Figure 44 Buildfile excluding FFLAGS statement.........111
Figure 45 Buildfile with options keyword 111
Figure 46 Array initialization error messages 162

Tables

Table 1 -no arguments 57
Table 2 -check option arguments 59
Table 3 -extend_dim effects on Fortran arrays (C Series only) 86
Table 4 -extend_dim effects on C arrays (C Series only) 86
Table 5 Levels of inlining 87
Table 6 -link_sort options 87
Table 7 phase arguments 88
Table 8 inline/no_inline statements and equivalents. 101
Table 9 clone/no_clone statements and equivalents. 102
Table 10 Overridable options statement options. 112
Table 11 Unsupported Fortran and C options. 171
Table 12 Compiler options to apc. 172
Table 13 Options that produce no executable 172
Table 14 Code generation options. 172
Table 15 -D option. 172

How to use this book

Purpose and audience

This guide has two purposes. First, it can be used as a text to introduce important concepts and teach you how to use the Application Compiler. Second, it provides experienced users with a guide to making the best use of the Application Compiler and getting the best performance out of their code.

This guide assumes that you are proficient in Fortran or C programming and in using the Convex Fortran or Convex C compiler. It also assumes you are familiar with the optimization concepts discussed in the *Fortran* or *C Optimization Guide (C Series)* or the *Exemplar Programming Guide.*

Organization

The chapters of this guide are loosely grouped into three parts. Chapters 1 through 6 describe the mechanics involved in using the Application Compiler. Chapter 7 gives a strategy for using the Application Compiler to optimize your programs. Chapters 8 and 9 provide hints for Fortran and C programmers who want to get the best performance out of their code. The appendixes include supplementary material covering compiler options, the PDBView utility, and the Application Compiler's error and warning messages.

Notational conventions

This guide uses the following notational conventions:

- The word "enter" in a phrase such as "enter a command" means that you type the command and press the carriage-return key. In contrast, the word "type" (for example, "type a line of text") means that you do not press the carriage-return key.
- *Italic* is used for the titles of documents.
- *Italic* is used to emphasize words of special importance, such as technical terms being used for the first time.

- `Constant-width font` is used to show output in examples and for "computer terms" in text. Computer terms include languages keywords, procedure and variable names, system calls, structures and data types, program statements, command names and options, directives, directories, and file names.
- In an example showing program output, *italics* indicate that the word enclosed is not a literal, but stands for some output that the program substitutes at runtime. Consider the following example:

  `Warning: file` *file_name* `is too long!`

  The italics show that the literal "*file_name*" does not appear in the output. Some specific file name appears there instead.
- Angle brackets are used to indicate that part of the output has been abbreviated for brevity. For example, the following line indicates the point where a compiler Optimization Report would be seen in the actual output:

  ```
  <Output Report>
  ```
- An ellipsis (. . .) in an example of code or program output shows that something, such as lines of code, has been left out to simplify the example.

Note

A Note highlights important supplemental information

Caution

A Caution highlights information necessary to avoid damage to equipment, software or data.

SPP Series only

Information in this paragraph applies to SPP Series machines only.

C Series only

Information in this paragraph applies to C Series machines only.

Associated documents

Using the Application Compiler can require information beyond the scope of this guide. The following documents can provide additional information to help you use this product successfully:

- The *Fortran Optimization Guide (C Series)* (DSW-034) provides background information on optimization concepts as they apply to the Fortran programming language running on C Series machines.
- The *Fortran User's Guide* (DSW-038) provide complete information on using the Convex Fortran language.

- The *C Optimization Guide (C Series)* (DSW-086) provides background information on optimization concepts as they apply to the C programming language running on C Series machines.
- The *C User's Guide* (DSW-086) provides complete information on using the Convex C language.
- The *Exemplar Programming Guide* (DSW-067) provides information on optimization and efficient programming techniques for Fortran and C running on Exemplar computers.
- The *American National Standard for Information Systems Programming Language* C (Document Number: X3J11/90-013) provides a complete description of ANSI Standard C.
- *C: A Reference Manual*, by Samuel P. Harbison and Guy L. Steele, Jr., (Prentice-Hall, Inc., 1987) provides a general reference to features of the C language.
- *The C Programming Language, Second Edition*, by Brian W. Kernighan and Dennis M. Ritchie (Prentice-Hall, Inc., 1988) is the classic work by the original developers of the C language.
- The *Performance Analyzer (CXpa) User's Guide* (DSW-251) explains how to profile your programs using the Convex Performance Analyzer.
- The *CXdb Concepts* (DSW-471), *CXdb User's Guide* (DSW-473), and *CXdb Reference* (DSW-472) manuals describe how to use the CXdb debugger.

Ordering documentation

To order the current edition of this or any other Convex document send requests to:

Convex Computer Corporation
Customer Service
P.O. Box 833851
Richardson, TX 75083-3851 USA

Include the order number or the exact title, as listed on the front cover.

Technical assistance

If you have questions that are not answered in this book, contact the Convex Technical Assistance Center (TAC). Use the phone numbers in the following table.

Location	Phone number

Within the continental U.S.
  Convex customers call
  Convex employees call
	
	(800) 952-0379
	(800) 545-4839

In Canada	(800) 345-2384
Outside continental U.S.	Contact local Convex office

Principles and concepts

1

Optimization modifies code to improve performance, while preserving output equivalent to that generated by the original code. In the early days of computing, programs were optimized by hand: programmers would go over the code line by line, looking for ways to speed things up. Today, optimizing compilers have replaced hand optimization for most large applications. The Convex Application Compiler represents the state of the art in optimizing compilers, providing maximum code optimization with a minimum of programmer intervention.

Basic optimization concepts

Before examining optimizations specific to the Application Compiler, you must understand basic optimization concepts as they apply to all Convex compilers.

Convex currently supports two supercomputer architectures: vector/parallel (C Series machines) and scalable parallel (SPP Series machines). C Series machines have from 1 to 8 processors, each of which is equipped with both scalar processing hardware and either 8 or 16 *vector registers*. A vector register can hold up to 128 64-bit elements; all of these elements can be manipulated by a single vector instruction (for instance, a value can be added to all 128 elements in a single operation). SPP Series machines have no vector registers, but instead contain from 2 to 128 single-chip scalar processors. These processors employ large local data caches which speed scalar processing by minimizing memory accesses.

While most optimizations performed by compilers running on these different architectures are conceptually similar, hardware differences require them to diverge in some areas. This means that while most code that can be optimized on one architecture can be optimized on the other, the actual optimizations performed will at least differ at the implementation level, and possibly also at the conceptual level.

In procedural Convex compilers, optimizations are grouped into levels; these levels of optimization can be specified via a command line option when the compiler is invoked (they can also be specified in the Application Compiler's buildfile as described in Chapter 3). The optimization options available are `-no`, `-O0`, `-O1`, `-O2`, and `-O3`. The optimizations performed become more aggressive (and thus generally provide higher performance) as the optimization level increases. Each successive level includes all of the optimizations performed by the levels preceding it.

The following sections discuss various optimization levels available in the procedural Convex compilers as they apply to each architecture. `-no`, the default optimization level on C Series machines only, is not covered because it disables all but the most basic machine-dependent scalar optimizations. The default optimization level on SPP Series machines is `-O2`.

For a detailed description of how the following optimizations apply to C Series machines, consult the *Fortran Optimization Guide (C Series)* or *C Optimization Guide (C Series)*. For a detailed description of how they apply to SPP Series machines, consult the *Exemplar Programming Guide*.

-O0 optimizations

The main goal of `-O0` optimizations is to create efficient scalar code at the basic-block level on both C Series and SPP Series architectures. A basic block is a sequence of statements ending with a conditional or unconditional branch. `-O0` optimizations are performed within a single basic block; they do not span basic blocks.

Basic block optimizations mostly involve making more efficient use of registers, eliminating code that never executes or is redundant, and efficiently scheduling assembly language instructions. Code is not moved at this level.

Optimizations performed at `-O0` include:

- Multiple-statement instruction scheduling
- Redundant-assignment elimination
- Assignment substitution
- Common-subexpression elimination
- Redundant-use elimination
- Constant propagation and folding
- Algebraic and trigonometric simplification

-O1 optimizations

The main goal of -O1 optimizations is to create efficient scalar code at the procedure level. A procedure is a function, subroutine or MAIN program in Fortran or a function in C. -O1 optimizations expand on -O0 optimizations by broadening their scope to cover entire procedures rather than just basic blocks, and add some new optimizations which involve limited code motion.

-O1 optimizations are performed on both C Series and SPP Series hardware platforms. They include:

- Constant propagation and folding
- Redundant-assignment elimination
- Dead-code elimination
- Hoisting and sinking scalar and array references from loops
- Copy propagation
- Common subexpression elimination
- Removal of invariant expressions from loops
- Strength reduction

-O2 optimizations

The main goal of -O2 optimizations, regardless of hardware platform, is to maximize data reuse once the data has been fetched from main memory, thus minimizing the number of main memory accesses necessary. This is accomplished by loading the data into some kind of processor-local memory, or *localizing* the data. The data in question is typically an array that is being manipulated by a loop.

On C Series machines, a vector register can often be used to manipulate such arrays; once the array (or a portion of it) is loaded, several operations can be performed on some or all of the elements in the vector register before they are stored back to main memory. This is called *vectorization.*

On SPP Series machines, the processor data cache is used similarly; the difference is that SPP Series machines must manipulate the array one element at a time in a scalar processor rather than many elements at a time in a vector register. Since the elements are not being simultaneously manipulated, efficient cache use on SPP Series machines depends on finding array sections that are heavily reused and keeping them in the cache.

On both platforms, the compiler can perform a number of transformations on loops to facilitate more efficient data localization. The most fundamental of these transformations is

strip mining. Strip mining breaks a loop that is too large to fit entirely into processor-local memory into a loop nest; the inner loop can then make efficient use of processor-local memory by either vectorization or, under certain conditions, caching, while the outer loop runs the inner loop as many times as necessary to cover the entire original trip count.

Data localization through vectorization can often be profitable for singly-nested loops on C Series machines; however, in order to efficiently use the processor data cache on SPP Series machines, multiple loop nests containing certain specific data reuse patterns are necessary. Refer to the *Exemplar Programming Guide* for more information on Exemplar-specific data localization optimizations.

Other loop transformations are performed at `-O2`; many of them either facilitate more efficient strip mining or are performed on strip mined loops to optimize processor-local memory usage. They include:

- Loop distribution
- Loop interchange
- Paired hoist and sink
- Redundant-test elimination
- Loop boundary-value peeling
- Loop blocking
- Loop unroll and jam
- Test promotion
- Pattern matching
- Scalar replacement

These optimizations are not profitable for all loops. Refer to the "Problems of optimization" section of this chapter for more information.

-O3 optimizations

The main goal of `-O3` optimizations, regardless of hardware platform, is to parallelize loops. Loops that have been data-localized are prime candidates for parallelization; individual iterations of inner loops that contain strips of localizable data can be parcelled out among several processors and run simultaneously. The maximum number of processors that can be used is limited by the number of iterations of the outer loop, and, of course, by processor availability.

While most parallelization is done on nested, data-localized loops, other code can also be parallelized. For example, through the use

of manually-inserted compiler directives, sections of code outside of loops can also be parallelized.

The Application Compiler

The procedural Convex C, Fortran, and Ada compilers use state-of-the-art compiler technology to produce highly optimized programs as good as, or better than, the best hand-optimized code. Over the years, these compilers have been improved to meet increasing demands for fast performance caused by growing problem size. Although new optimizations continue to be added, these compilers have been developed to the point that further incremental changes can no longer be expected to produce major improvements in performance. To meet the performance needs of tomorrow's programs, a new approach is needed.

The procedural Convex compilers can perform all of the optimizations discussed in the "Basic optimization concepts" section of this chapter. Optimization is inhibited on any hardware platform, however, when the compiler lacks the information it requires to weigh the safety of a particular optimization. This often happens when code in one procedure is dependent on, or affected by, code in another procedure. Most compilers treat each procedure separately. Information that is local to other procedures is not available to the compiler. Optimizations performed outside the scope of a basic block are sometimes called "global," but they are still confined in scope to a single procedure.

For Convex Fortran and Convex C programs, the Convex Application Compiler goes beyond global optimization. The Application Compiler contains a new component: an interprocedural analyzer that tracks the flow of data and control between procedures. The information generated by this analysis removes the scope restrictions on optimization, which allows the Application Compiler to generate more efficient code by taking the entire program, with all of its dependencies, into account. The database of program information that the interprocedural analyzer builds up also allows the Application Compiler to perform better error checking, leading to more robust and reliable programs. Finally, the Application Compiler saves development time by automatically performing certain optimizations, such as procedure inlining, that previously had to be performed by hand.

Application compiling

Figure 1 shows the internal operation of the Application Compiler. User-visible components (the buildfile and `apc` command) are shown on top. `apc` calls several other programs, shown in the bottom row, to create an executable.

`apc` controls the entire compile process. Like `make`, `apc` uses a text file created by the user. In the case of `make`, the text file, called a makefile, specifies the dependencies between source files that require compiles to be performed in a specific order. `apc` gets the dependency information automatically by scanning the target application's source files. Buildfiles therefore tend to be simpler and easier to write than makefiles; often, they include only the list of options that the programmer wants to use in compiling the target application.

Unlike `make`, which is simply a utility for automating compilations, `apc` is the visible front end to the Application Compiler. Therefore, you cannot invoke the Application Compiler without using `apc`.

The Application Compiler includes the functionality of the Convex C and Fortran compilers (shown as the summary step in Figure 1 and Figure 2). Figure 1 shows how `apc`, relying on the buildfile to provide necessary instructions, controls each phase of program optimization. In this and all following illustrations, boxes represent processes or phases and ellipses represent files or data.

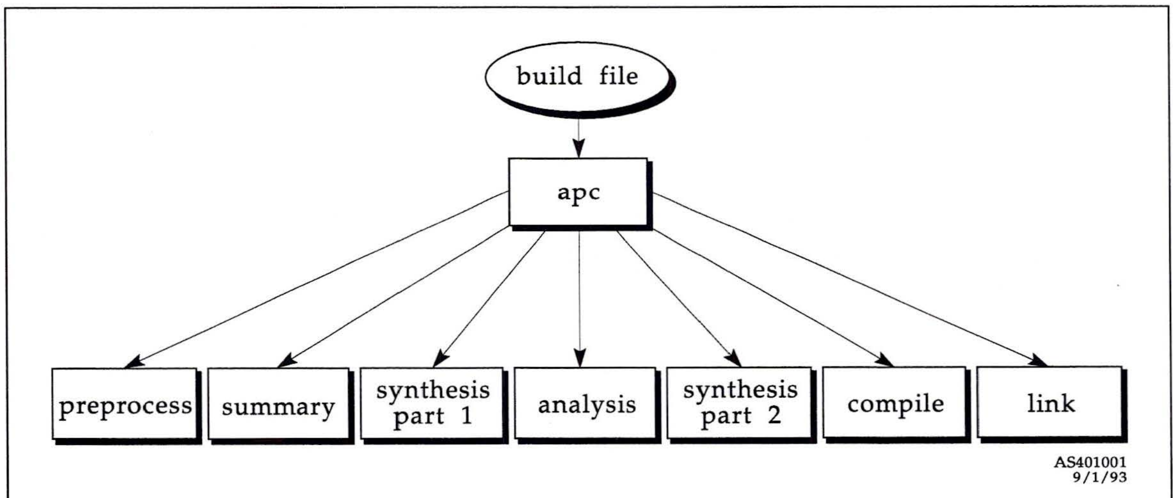


Figure 1 Control flow

Application compilation occurs in several phases, as shown in Figure 1. The preprocess phase is identical to the preprocess phase of the procedural compilers; in fact, the same preprocessors that are used with the conventional compilers are used with the APC.

Optimization occurs in the middle 5 phases shown in Figure 1. Between phases, an intermediate representation of the program is stored in a program database. This intermediate representation is examined by some phases and manipulated by others.

In the summary phase, the Application Compiler analyzes each procedure separately. During this phase, the Application Compiler generates the intermediate-code representation, performs scalar optimizations on it, and stores it in the program database. This phase employs the procedural compiler front ends.

The synthesis phase part 1, analysis phase, and synthesis phase part 2 all examine the intermediate representation of the program and generate optimization information which is stored in the program database. These phases have no analogues in the procedural compilers.

This information is then used by the compile phase to optimize the internal representation of the program and produce object code, which is linked by the standard linker (via `fc` or `cc`) to create an executable. The compile phase employs the procedural compiler back ends.

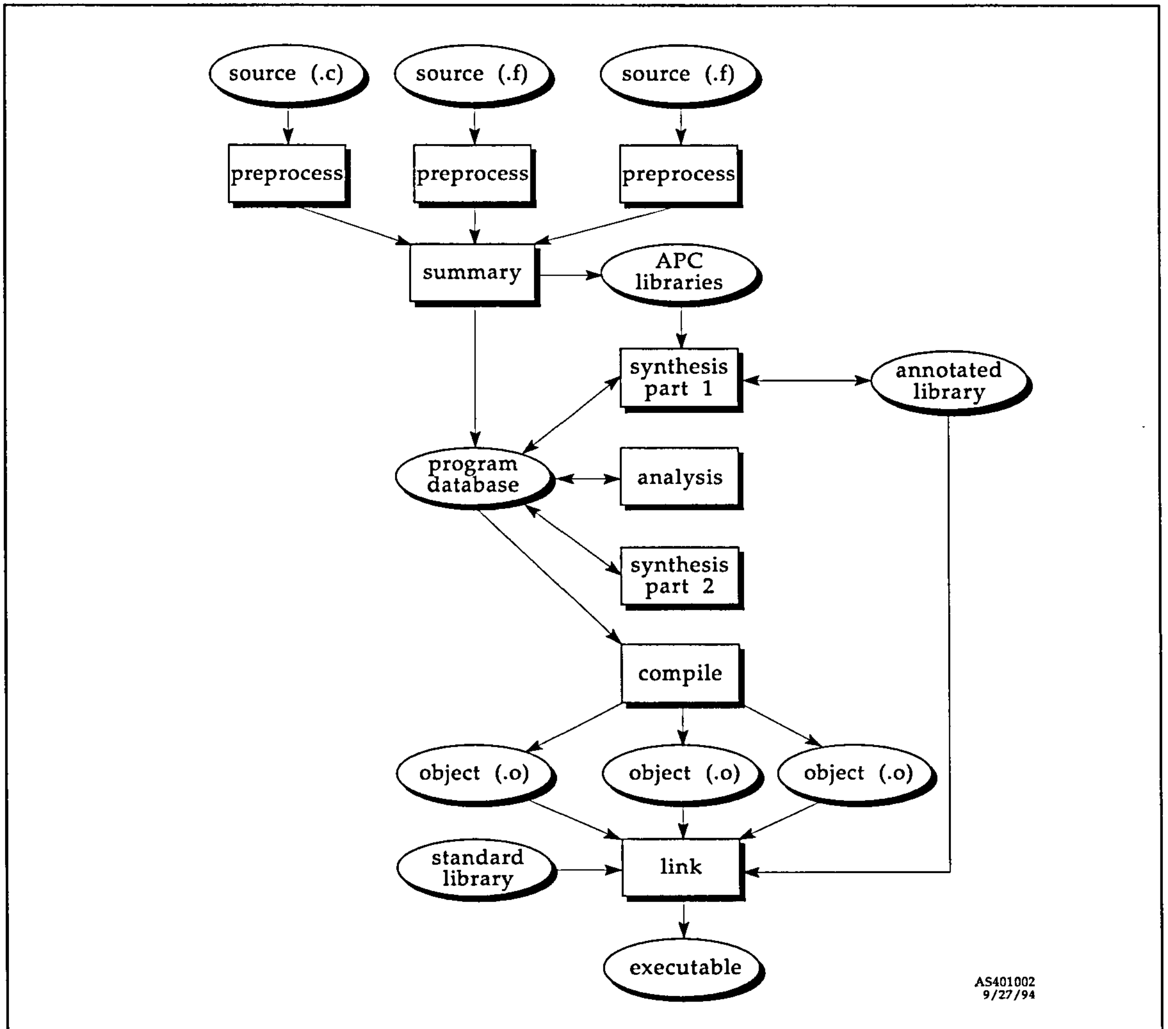


Figure 2 Data flow

Figure 2 shows three types of libraries that can be used in a program. Procedural object-code libraries, shown in the lower left, do not contain the necessary information to allow for interprocedural optimization. The Application Compiler can still optimize a program that uses such libraries, but the effectiveness of interprocedural optimization is drastically reduced because the analyzer does not know what is going on in the library procedures. This forces the Application Compiler, like the procedural compiler, to make more conservative assumptions in such cases.

The second type of library is associated with a separate annotation file. The annotations contained in this file are converted into

database entries for the library procedures by the Application Compiler. These annotations include the side effects of each procedure and allow more complete program optimization.

The third type of library, called an APC library, contains supplementary optimization information. These libraries are stored in intermediate representation format rather than as finished object code. APC libraries allow the highest level of optimization.

Figure 3 shows the summary phase in greater detail. The Application Compiler scans the source code and creates an intermediate representation (IR), which is stored in the program database, as is the buildfile. The apc command, which controls the analysis, communicates with the compiler directly and through the database.

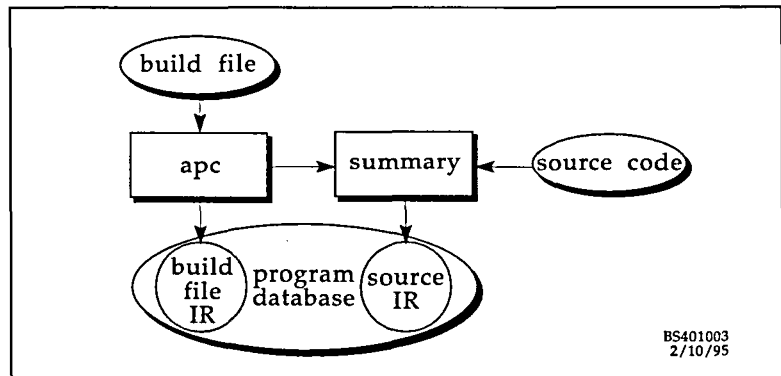


Figure 3 Summary phase

Physically, the program database is a directory named PDB that the Application Compiler creates in the directory from which apc was invoked. The -path option to apc allows you to specify an alternate directory for the PDB directory, as explained under "Program database options" in Chapter 3.

The PDB directory contains:

- a file named LOG, which contains all error and information messages generated by the APC as well as a header describing the APC version and any command line options used.
- a directory named dskh containing one or more files suffixed with ".dskh"; these are binary database files.
- if the -library option is used to create an APC library, there will be one or more files suffixed with ".ir"; these are the intermediate representation files shown in Figure 3 .
- object files for each routine that was successfully compiled; these are suffixed with ".o".

- on SPP Series machines, assembly language files for each routine that was successfully compiled; these are suffixed with "`.s`".

Information contained in the program database can be viewed and analyzed using the PDB Viewer, as described in Appendix B.

As shown in Figure 4, the next three phases collect information from the intermediate representations stored in the program database by the summary phase. Part 1 of the synthesis phase gathers optimization information for procedure calls and variables that are passed by reference, and stores it in the program database. By default, compilation will terminate with an error message if unresolved symbols are detected in this phase; this can be overridden using the `-permit unresolved` command line option. If this option is used and unresolved symbols are detected, a warning is issued and compilation continues to the link phase, where it terminates. Refer to Chapter 3 for more information.

After part 1 of the syntheses phase, the analysis phase performs another scalar optimization analysis, using information generated in part 1 of the synthesis phase, and stores its results in the program database.

Part 2 of the synthesis phase performs interprocedural optimization analysis and gathers information on procedure side effects, storing its results in the program database.

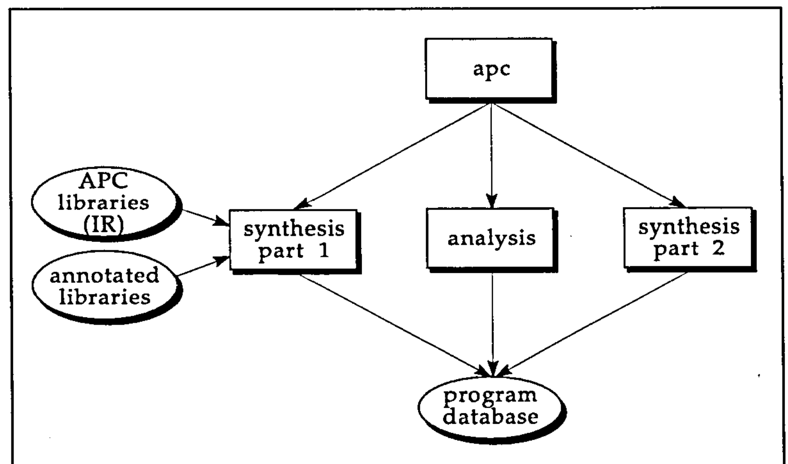


Figure 4 Synthesis/analysis/synthesis phases

Finally, all the optimization information generated in the synthesis/analysis phases is incorporated into the intermediate representation of the program and the compile phase executes, implementing all the optimizations that up until now have only been noted in the database. This phase is analogous to the

back-end phase of the procedural compiler, and produces object code. See Figure 5.

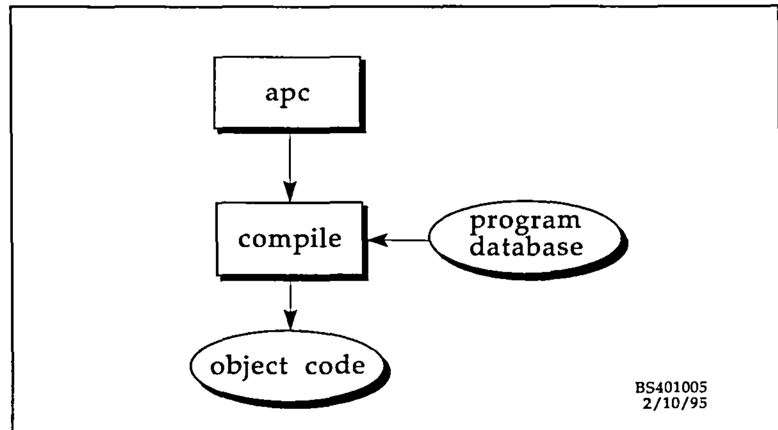


Figure 5 Compile phase

This object code is then linked with both APC-annotated and non-APC libraries that are used in the program to produce an executable, as shown in Figure 6.

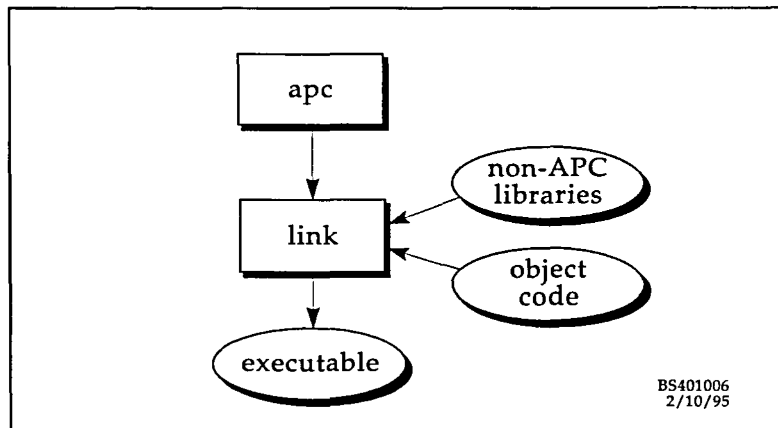


Figure 6 Link phase

The summary, synthesis part 1, analysis, synthesis part 2, and compilation phases are transparent and inaccessible to you. You can invoke `apc` directly or from a makefile. A makefile can generate part of your program's source, then invoke `apc`. For example, a makefile might call `lex` to generate a scanner for the front end of a program, then call `apc` to compile and optimize the entire program. The `make` utility can pass appropriate option flags from its own command line to `apc`. You can set default options within the makefile.

Some vendors have introduced interprocedural optimizers that require the user to work interactively with the compiler. These optimizers require the user to identify constructs that are safe candidates for interprocedural optimization. In a complex program where the data paths may not be obvious, this is not always possible. Moreover, the interactive nature of the optimization makes compile times extremely slow and prone to error.

Although interprocedural analysis is necessarily slower than normal procedural or "global" optimization, the Application Compiler's automatic approach still achieves reasonable compile times. To further speed compilations, the Application Compiler retains the program database so that the entire program need not be reanalyzed when a single procedure is changed.

Because the database uses a fair amount of disk space, the `-c` option is provided to remove the program database when you no longer need it.

Problems of optimization

Optimizing code can dramatically improve a program's performance. As described in the "Basic optimizations" section, the most beneficial optimizations, regardless of hardware platform, are those that facilitate data localization. On C Series machines this is accomplished through vectorization, which can often reduce a program's runtime by as much as a factor of ten; on multi-headed vector machines, runtimes can be reduced even further (up to a factor of N for an N-processor machine) if the programs use appropriate algorithms.

Certain loops that vectorize on vector machines can realize similar data-localization speedups on scalable parallel machines, where they are optimized to run scalar on several fast scalar processors. Again, a speedup factor of N for an N-processor machine is theoretically possible.

Some constructs, however, present problems that can prevent a program from realizing the full benefits of optimization. These problems, which can be reduced through interprocedural optimization, include

- Code that does not vectorize or parallelize
- Optimized code that gives wrong answers, due to aliases or other coding errors
- Optimized code that runs slower than the original code

The most frequent cause of the compiler's failure to vectorize loops on a C Series machine is a *recurrence* (recurrences do not necessarily prevent data localization on SPP Series machines, but

often restrict parallelization). A recurrence exists when data created in one iteration of a loop is referenced on a subsequent iteration. Consider the loops shown in Figure 7 and Figure 8.

The loop in Figure 7 is free of recurrences and, on a C Series machine, can be vectorized without difficulty. The arrows in this figure show the calculation and assignment of individual array elements. Note that the loops formed by these arrows do not meet one another. The computation of each array element is *independent* of all other calculations in the loop.

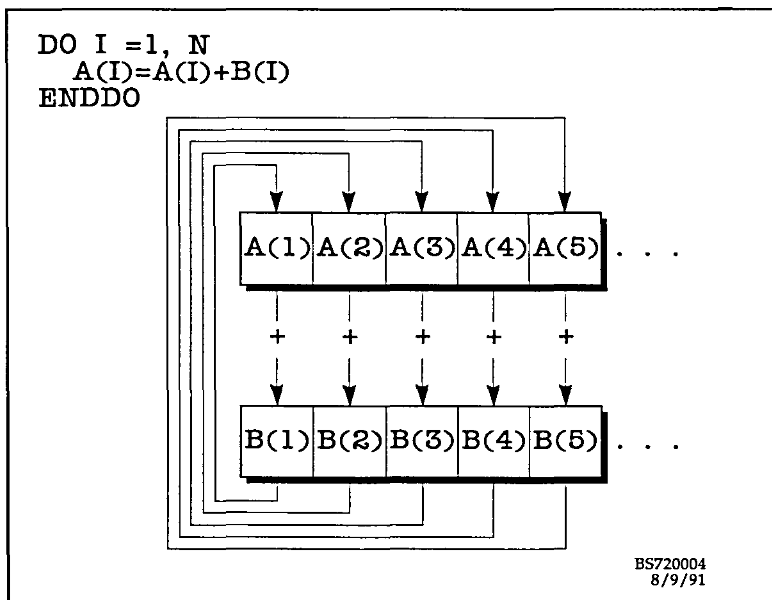


Figure 7 Loop without recurrence

This data independence means that it is safe to break A up into sections to facilitate vectorization on C Series machines. This allows the computation of several array elements simultaneously through use of a vector processor or multiple CPUs operating in parallel.

Now look at Figure 8. This loop has a recurrence caused by the reference to `A(I-1)`. Note how the flow of calculations proceeds in a spiral fashion with the output from each calculation feeding into the next. The value computed on the first iteration is used on the second iteration, and so on for the duration of the loop. Each calculation is dependent on the previous calculation.

This data dependency, or recurrence, prevents vectorization and/or parallelization. (For a more complete discussion of recurrence and its impact on vectorization, see the *Convex Fortran* or *Convex C Optimization Guide (C Series)*; for a discussion of how

data dependencies impact SPP Series machines, see the *Exemplar Programming Guide.*)

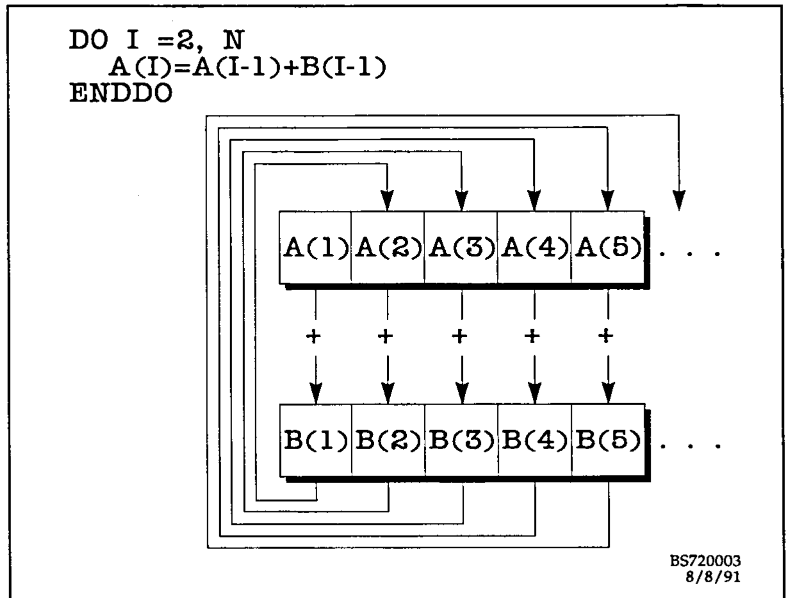


Figure 8 Loop with recurrence

If a genuine recurrence exists, no optimizer or compiler can remove it without adversely affecting the results. Often, however, a compiler simply lacks sufficient information to tell whether or not a recurrence exists. In these situations the compiler assumes an *apparent recurrence.* Programmers can use NO_RECURRENCE or FORCE_PARALLEL directives with C Series compilers or NO_DEPENDENCE or LOOP_PARALLEL with SPP Series compilers to instruct the compiler to ignore an apparent recurrence. However, if any of these directives is mistakenly used on a loop that contains a real recurrence or dependence, errors can result.

Through interprocedural analysis, the Application Compiler can often distinguish between real and apparent recurrences. As a result, many loops that cannot be vectorized or parallelized with the procedural compilers can be when compiled with the Application Compiler.

Often, the procedural compiler cannot vectorize a loop because of an embedded procedure call. The Application Compiler can inline many of these calls and thus increase the number of loops that can be vectorized on C Series machines. Similarly, inlining can increase the number of loops that can be blocked and/or parallelized on SPP Series machines. On both platforms, the Application Compiler can sometimes parallelize the loop with the

call intact, as described in the "Automatic parallelization of loops with calls" section of Chapter 2.

Wrong answers due to optimization are usually caused by aliases, accesses that go beyond the bounds of an array, or nonstandard programming practices. Through enhanced error checking, the Application Compiler can detect many of these errors.

Basic interprocedural optimizations

2

The Application Compiler (also referred to as the APC) performs the following basic interprocedural optimizations:

- Interprocedural constant propagation
- Automatic inlining
- Procedure cloning
- Automatic parallelization of loops with calls
- Enhanced error checking
- Alias and pointer tracking
- Link optimization
- Bank-conflict reduction

These optimizations are discussed in detail in the sections that follow.

Interprocedural constant propagation

When a constant value is assigned to a variable, subsequent references to that variable can be replaced by the constant value itself. This is called *constant propagation*. When a procedure is compiled at optimization level `-O0`, the procedural Convex C and Fortran compilers propagate constants within the scope of a basic block. When a procedure is compiled at optimization level `-O1` or above, these compilers propagate constants within the scope of an individual procedure. When a program is compiled at `-O1` or above using the Application Compiler, constants are propagated within and between procedures.

In the following example, the Fortran compiler propagates the constant value 128, which is assigned to the variable `N` in the original code. The compiler retains the constant and, until the time when a new value is assigned to `N`, it can substitute the constant for each reference to the variable `N`.

Because N is not used elsewhere in the procedure, constant propagation also allows the compiler to perform another optimization: it eliminates the assignment to N. Constant propagation improves performance because operations on constants are generally faster than operations on variables. Eliminating an assignment eliminates the need for a memory access, further improving performance.

Strip mining is necessary on C Series machines when the number of iterations of a loop might exceed the vector register length of 128. When the iteration count of a loop is variable, the compiler must assume that the loop will produce multiple strips when strip mined. When the code in the following example is compiled with the Application Compiler at -O2 on a C Series machine, constant propagation allows the compiler to vectorize the DO loop as a single strip. This minimizes the overhead associated with the vector loop.

Original code	Optimized code

SUBROUTINE SB(AR)
REAL AR(*)
INTEGER N
 	
N=128
 	
DO I=1,N
  AR(I)=AR(I)/3.1416
ENDDO
END
	SUBROUTINE SB(AR)
	REAL AR(*)
	INTEGER N
	
	!Assignment eliminated
	
	DO I=1,128
	  AR(I)=AR(I)/3.1416
	ENDDO
	END

Interprocedural constant propagation can yield analogous gains on SPP Series machines in nested loops that are candidates for loop blocking. In this case, propagated constants can be used to determine more efficient blocking factors for the loops. For more information on loop blocking, see the *Exemplar Programming Guide.*

The Application Compiler propagates constants between procedures as well as within them. The following example shows how the APC propagates a constant, 128, between procedures.

Original code	Optimized code

PROGRAM MAIN
REAL AR(128)
 	
N=128
CALL SB(AR,N)
END
 	
SUBROUTINE SB(AR,N)
REAL AR(*)
INTEGER N
 	
DO I=1,N
  AR(I)=AR(I)/3.1416
ENDDO
END
	PROGRAM MAIN
	REAL AR(128)
	
	N=128
	CALL SB(AR,N)
	END
	
	SUBROUTINE SB(AR,N)
	REAL AR(*)
	INTEGER N !N=128
	
	DO I=1,128
	  AR(I)=AR(I)/3.1416
	ENDDO
	END

In the original code, the variable N, passed in from the MAIN section, determines the iteration value of the loop in subroutine SB. Because the Application Compiler analyzes the program as a whole, it knows that the value of N passed to SB is a constant. The APC propagates the constant from MAIN into subroutine SB.

Just as in the previous example, constant propagation causes the compiler to vectorize the loop on C Series machines without any strip-mine overhead. The procedural compilers would generate a strip-mined vector loop in the example above. To manually eliminate the strip-mine overhead, you would have to look at the code, realize what was happening, and insert a directive before the loop.

Again, similar gains are realized on SPP Series machines in loops that can be blocked.

Apparent recurrences

The propagation of constants between procedures allows the Application Compiler to resolve apparent recurrences in many loops. The following code contains an apparent recurrence on the I loop in subroutine SB1.

Original code	Optimized code
PROGRAM MAIN	PROGRAM MAIN
PARAMETER(M=200)	PARAMETER(M=200)
REAL U(500)	REAL U(500)
CALL SB1(U,M)	CALL SB1(U,M)
END	END
SUBROUTINE SB1(U,M)	SUBROUTINE SB1(U,M)
REAL U(*)	REAL U(*)
INTEGER M	INTEGER M ! M=200
DO I=1,100	DO I=1,100
U(I)=U(I+M)*4.0	U(I)=U(I+200)*4.0
ENDDO	ENDDO
END	END

Optimizing one procedure at a time, the procedural compiler does not know what values other procedures pass to SB1. This uncertainty creates an apparent recurrence on the I loop in SB1. If the value of M is negative, the loop assigns a value on one iteration to an array location that is accessed on a future iteration, and a recurrence exists. The Application Compiler propagates the value of the PARAMETER constant M into subroutine SB1. The APC now knows that the value added to I is positive, so no recurrence exists, and it is safe to vectorize the loop on a C Series machine.

Loop strides

To vectorize a loop, the compiler must determine whether the loop's stride (the value by which the iteration variable changes on each iteration) produces a potential recurrence. Interprocedural constant propagation allows the Application Compiler to determine many loop strides that would not be determinable by the procedural compilers. In the following example, the procedural compiler does not know the value of N, which determines the stride of the loop in SB2.

Original code	Optimized code

PROGRAM MAIN
 	
PARAMETER(L=1)
PARAMETER(M=200)
PARAMETER(N=10)
REAL A(500),B(500)
 	
CALL SB2(A,B,L,M,N)
END
 	
SUBROUTINE
SB2(A,B,L,M,N)
 	
REAL A(*),B(*)
INTEGER L,M,N
 	
DO I=L,M,N
  A(I)=A(I+1)+B(I)
ENDDO
END
	PROGRAM MAIN
	
	PARAMETER(L=1)
	PARAMETER(M=200)
	PARAMETER(N=10)
	REAL A(500),B(500)
	
	CALL SB2(A,B,L,M,N)
	END
	
	SUBROUTINE
	SB2(A,B,L,M,N)
	
	REAL A(*),B(*)
	INTEGER L,M,N
	!L=1,M=200,N=10
	DO I=1,200,10
	  A(I)=A(I+1)+B(I)
	ENDDO
	END

If the stride of the loop, determined by the value of `N`, is negative, the `I` loop steps through the array backwards, causing a potential recurrence between `A(I)` and `A(I+1)`. The Application Compiler propagates the constant value assigned to `N` by `MAIN` into subroutine `SB2`. The stride of the loop is known to be positive, and the loop can be vectorized.

Data dependencies can similarly prevent data reuse by certain nested loop constructs on SPP Series machines, and can be similarly eliminated through interprocedural constant propagation.

Loop interchange

Propagating constants between procedures enables the Application Compiler to interchange loops in a more intelligent manner. Consider the following example.

Original code	Constant propagation
PROGRAM MAIN	PROGRAM MAIN
PARAMETER(N=8,M=200)	PARAMETER(N=8,M=200)
REAL A(N,M),B(N,M)	REAL A(N,M),B(N,M)
REAL C(N,M)	REAL C(N,M)
CALL S3(A,B,C,M,N)	CALL S3(A,B,C,M,N)
END	END
SUBROUTINE	SUBROUTINE
S3(A,B,C,M,N)	S3(A,B,C,M,N)
REAL A(N,M),B(N,M)	REAL A(8,200),B(8,200)
REAL C(N,M)	REAL C(8,200)
INTEGER N,M	INTEGER N,M !N=8,M=200
DO J=1,N	DO J=1,8
DO I=1,M	DO I=1,200
A(J,I)=B(J,I)+C(J,I)	A(J,I)=B(J,I)+C(J,I)
ENDDO	ENDDO
ENDDO	ENDDO
END	END

The procedural compiler would interchange the loops in S3 because the loops are not written to access memory in the most efficient manner. The APC propagates the values of N and M into S3. It then knows that the I loop has a much larger iteration count than the J loop. It does not interchange the two loops. This sacrifices efficient memory accesses, but more than makes up for it by preserving the more efficient strip length on the inner loop. This represents a gain on both hardware platforms.

All the examples up to this point show the propagation of constants from a higher-level procedure (the caller) to a lower-level procedure (the callee). The Application Compiler can also propagate constants assigned in other procedures anywhere in the call tree, as well as constants assigned to Fortran COMMON blocks and C global variables.

The following example is similar to the previous one. Instead of being assigned a constant value in MAIN, M and N are part of a COMMON block assigned in subroutine INIT. Here again, the APC propagates the constant value between calls and correctly handles the loop interchange.

Original code	Constant propagation
PROGRAM MAIN	PROGRAM MAIN
REAL A(8,200),B(8,200) REAL C(8,200)	REAL A(8,200),B(8,200) REAL C(8,200)
CALL INIT CALL S4(A,B,C) END	CALL INIT CALL S4(A,B,C) END
SUBROUTINE INIT COMMON N,M N=8 M=200 END	SUBROUTINE INIT COMMON N,M N=8 M=200 END
SUBROUTINE S4(A,B,C)	SUBROUTINE S4(A,B,C)
REAL A(8,200),B(8,200) REAL C(8,200) COMMON N,M	REAL A(8,200),B(8,200) REAL C(8,200) COMMON N,M !N=8,M=200

DO J=1,N
  DO I=1,M
   A(J,I)=B(J,I)+C(J,I)
  ENDDO
ENDDO
END
	DO J=1,8
	  DO I=1,200
	   A(J,I)=B(J,I)+C(J,I)
	  ENDDO
	ENDDO
	END

Dead code elimination

Propagating constants between procedures can also allow the Application Compiler to eliminate unreachable or "dead" code. Dead code includes the unreachable alternatives of a conditional statement embedded within a loop. By eliminating such conditionals, the APC increases the probability that loop data can be localized, or can improve the efficiency of a loop that already employs data localization. In the following example, the APC propagates the constant value assigned to N into subroutine SB5. By substituting this value for N, the APC eliminates the conditional test embedded in the DO loop, resulting in more efficient vectorization on C Series machines.

Original code	Optimized code

PROGRAM MAIN
 	
REAL A(500),B(500)
INTEGER N, P
 	
P=50
N=P*2 !N=100
 	
CALL SB5(A,B,N)
...
END
 	
SUBROUTINE SB5(A,B,N)
 	
REAL A(*),B(*)
INTEGER N
 	
DO I= 1,N
  IF (N.LT.100) THEN
    A(I)=A(I)-B(I)*2.0
  ELSE
    B(I)=B(I)-A(I)*2.0
  ENDIF
ENDDO
END
	PROGRAM MAIN
	
	REAL A(500),B(500)
	INTEGER N, P
	
	P=50
	N=100
	
	CALL SB5(A,B,N)
	...
	END
	
	SUBROUTINE SB5(A,B,N)
	
	REAL A(*),B(*)
	INTEGER N ! N=100
	
	DO I= 1,100
	  B(I)= B(I)-A(I)*2.0
	ENDDO
	END

On SPP Series machines, where such loops must run scalar, eliminating such conditionals speeds up execution. Conditional elimination can also contribute to other optimizations on SPP Series machines.

Summary

To summarize, the Application Compiler propagates constants assigned to variables that are passed as C parameters or Fortran arguments or transmitted as C globals or Fortran COMMON variables. It does not propagate constants returned by functions.

Constants can be propagated from a higher-level procedure (a caller), between procedures at the same level, or from a lower-level procedure to a higher-level procedure. Constants can be propagated down through any number of levels of procedure calls. In many cases, propagation of constants between procedures enables the APC to resolve recurrences, determine loop strides,

interchange loops more intelligently, omit unnecessary strip mines, and eliminate unnecessary code.

Automatic inlining

Inline substitution, or inlining, replaces a procedure call with a copy of the code that makes up the body of the called procedure. When a procedure is inlined, the Application Compiler replaces formal parameters or dummy arguments with the corresponding actual parameters or arguments.

The following example shows how the APC inlines a Fortran subroutine. Note that the actual parameter `A` replaces the formal parameter `B` in the inlined code.

Original code	Inlined code

<pre>PROGRAM MAIN
REAL A (60,4)
DO I=1,4
  DO J=1,60
    A(J,I)=J*I
  ENDDO
ENDDO
CALL SUBA(A)
END
 	
SUBROUTINE SUBA(B)
REAL B(60,4)
PRINT *,B(1,1)
RETURN
END</pre>
	<pre>PROGRAM MAIN
	REAL A (60,4)
	DO I=1,4
	  DO J=1,60
	    A(J,I)=J*I
	  ENDDO
	ENDDO
	PRINT *,A(1,1)
	END</pre>

Inlining eliminates procedure-call overhead and allows additional optimizations. Some Fortran procedures can be inlined semiautomatically without using the Application Compiler, but inlining of Convex C procedures is only available via the APC. Not only does the APC save time that you would otherwise spend preparing files for inlining, it analyzes profitability and determines which procedures to inline. (You can modify or override this analysis using command line options or directives.)

Improved data localization

Often the Application Compiler can perform additional optimizations once it has inlined a procedure. In this manner, interprocedural analysis improves the efficiency of ordinary, procedure-level optimizations.

For example, the following code has a loop in SUBA that cannot be vectorized without inlining on C Series machines. (The compiler does not know the possible values of N and thus cannot ensure that the loop is free of recurrences.) Once the APC inlines the procedure, as shown on the right, it can see that N is always positive and no recurrence exists. It can therefore localize the arrays A and B.

<table>
<tr><th>Original code</th><th>Inlined code</th></tr>
<tr><td><pre>PROGRAM MAIN
REAL A(100000), B(500)
INTEGER N

CALL INIT(A)
CALL INIT(B)

DO N=1,500
  CALL SUBA(A,B,N)
ENDDO
END

SUBROUTINE SUBA(A,B,N)
REAL A(100000), B(500)
INTEGER N

DO I=1,99500
  A(I)=A(I+N)*B(N)
ENDDO
END</pre></td><td><pre>PROGRAM MAIN
REAL A(100000)
REAL B(500)
INTEGER N

CALL INIT(A)
CALL INIT(B)

DO N=1,500
  DO I=1,99500
    A(I)=A(I+N)*B(N)
  ENDDO
ENDDO
END</pre></td></tr>
</table>

On C Series machines, the loop nest that results from inlining can be strip mined and vectorized. On SPP Series machines, the loop nest can be blocked to insure that arrays A and B make efficient use of the processor data cache.

Improved loop parallelization

The Application Compiler is capable of automatically parallelizing loops containing calls as described in the "Automatic parallelization of loops with calls" section. However, under some circumstances, inlining procedure calls and parallelizing the loop containing the inlined call is preferable. For example, for small procedures the overhead of parallelizing the call may outweigh the performance gain; and for self-dependent procedures (refer to the "Self-dependence" section), parallelization of the containing loop is impossible without manual intervention.

In the following example, the function F1 is inlined; the resulting DO loop can then be easily and profitably parallelized.

Original code	Inlined code

PROGRAM MAIN
REAL A(10000), B(10000)
INTEGER N
 	
CALL INIT(A)
CALL INIT(B)
 	
DO N=1,10000
  A(N)=F1(A,B,N)
ENDDO
END
 	
REAL FUNCTION F1(A,B,N)
REAL A(10000), B(10000)
INTEGER I
F1 = A(N) + B(N)*N
END
	PROGRAM MAIN
	REAL A(10000), B(10000)
	INTEGER N
	
	CALL INIT(A)
	CALL INIT(B)
	
	DO N=1,10000
	  F1$1 = A(N) + B(N)*N
	  A(N)= F1$1
	ENDDO
	END

Dead code elimination

In some instances, inlining allows the Application Compiler to eliminate dead code. In the following example, SUBC contains an IF test that always fails. Without inlining, the compiler cannot tell this. Because SUBC inlines, the APC recognizes the first alternative as dead code and eliminates it.

Original code

```
PROGRAM MAIN
REAL A(1000)
REAL B(1000)
INTEGER N

CALL INIT(A)
CALL INIT(B)

DO N=10,100,10
  CALL SUBC(A,B,N)
ENDDO
END

SUBROUTINE SUBC(X,Y,N)
REAL X(1000)
REAL Y(1000)
INTEGER N
IF (N .LT. 10) THEN
  DO I=1,1000-N,N
    X(I) = Y(I+N)
  ENDDO
ELSE
  DO I=1,1000-N
    X(I) = X(I+N)
  ENDDO
ENDIF
END
```

With inlining

```
PROGRAM MAIN
REAL A(1000)
REAL B(1000)
INTEGER N

CALL INIT(A)
CALL INIT(B)

DO N=10,100,10
  IF (N.LT.10) THEN
    DO I=1,1000-N,N
      A(I) = B(I+N)
    ENDDO
  ELSE
    DO I=1,1000-N
      A(I) = A(I+N)
    ENDDO
  ENDIF
ENDDO
END
```

Optimized code

```
PROGRAM MAIN
REAL A(1000)
REAL B(1000)
INTEGER N

CALL INIT(A)
CALL INIT(B)

DO N=10,100,10
  DO I=1,1000-N
    A(I) = A(I+N)
  ENDDO
ENDDO
END
```

Procedure cloning

The procedural Convex Fortran compiler performs inlining and constant propagation within certain limits. Cloning, on the other hand, is a new optimization provided only by the Application Compiler. A *clone* is a duplicate of a procedure. The APC creates clones so that it can optimize the clone differently from the way it optimized the original.

The following example shows the cloning of a Fortran subroutine.

Original code	Optimized code

`DO N=1,1000`
`  CALL CPDAG(AR, MA, N)`
`  DO I=1,1000`
`    MD(N,I)=AR(I)`
`  ENDDO`
`ENDDO`
 	
`DO N=1,1000`
`  CALL CPDAG(AR, MD, 1)`
`  CALL MDDAG(MD)`
`ENDDO`
`...`
`SUBROUTINE CPDAG(AR, MX, N)`
`REAL AR(1000)`
`REAL MX(1000,1000)`
`INTEGER N`
 	
`IF (N.GT.0) THEN`
`  DO I=1,1001-N`
`    AR(I)=MX(I,I-1+N)`
`  ENDDO`
`ELSE`
`  DO I=1,1000+N`
`    AR(I)=MX(I-N,I)`
`  ENDDO`
`ENDIF`
`END`
	`DO N=1,1000`
	`  CALL CPDAG(AR, MA, N)`
	`  DO I=1,1000`
	`    MD(N,I)=AR(I)`
	`  ENDDO`
	`ENDDO`
	
	`DO N=1,1000`
	`  CALL CPDAG$CLONE$1(AR, MD, 1)`
	`  CALL MDDAG(MD)`
	`ENDDO`
	`...`
	`SUBROUTINE CPDAG(AR, MX, N)`
	`REAL AR(1000)`
	`REAL MX(1000,1000)`
	`INTEGER N`
	
	`IF (N.GT.0) THEN`
	`  DO I=1,1001-N`
	`    AR(I)=MX(I,I-1+N)`
	`  ENDDO`
	`ELSE`
	`  DO I=1,1000+N`
	`    AR(I)=MX(I-N,I)`
	`  ENDDO`
	`ENDIF`
	`END`
	
	`SUBROUTINE CPDAG$CLONE$1(AR, MX, N)`
	`REAL AR(1000)`
	`REAL MX(1000,1000)`
	`INTEGER N   ! N=1`
	
	`DO I=1,1000`
	`  AR(I)=MX(I,I)`
	`ENDDO`
	`END`

Subroutine CPDAG is called 2000 times in this example. The Application Compiler sees that for 1000 of these calls, the value passed to the dummy argument N in CPADG is 1. For the other 1000 calls, the value depends on some other calculations in the program that cannot be evaluated at compile time. The APC creates a duplicate of CPDAG, called CPDAG$CLONE$1 in this example, and substitutes a call to CPDAG$CLONE$1 for one of the

calls to CPDAG. It then optimizes CPDAG$CLONE$1 by propagating the constant 1 into the clone, which in turn allows the elimination of an N test and unreachable code in CPDAG$CLONE$1.

As this example shows, one optimization often leads to several others. This is one reason why the APC is so effective: the handful of new optimizations it adds greatly increases the number of situations where the previously available optimizations can be applied.

The APC can create clones and propagate constants to improve optimization opportunities, as shown in the previous example, or to simplify IF statements, as shown below. In the following example, the main program passes one of three constants to subroutine HAL. Cloning HAL allows the APC to propagate each of these constants. It then eliminates the unreachable ("dead") code from each of the clones and the original, allowing faster execution.

Original code	Optimized code

<pre>IF (SWITCH.EQ.0) THEN
  CALL HAL(0,R)
ELSE
  IF (SWITCH.EQ.1) THEN
    CALL HAL(1,R)
  ELSE
    CALL HAL(9000,R)
  ENDIF
ENDIF
...
SUBROUTINE HAL(NA,RI)
INTEGER NA
REAL RI(*)
IF (NA.GT.1) THEN
  DO I=1,7000
    RI(I)=1.0/I
  ENDDO
ELSE
  IF (NA.EQ.1) THEN
    RI(1)=1.0
  ELSE
    RI(1)=0.0
  ENDIF
ENDIF
END
...</pre>
	<pre>IF (SWITCH.EQ.0) THEN
	  CALL HAL(0,R)
	ELSE
	  IF (SWITCH .EQ. 1) THEN
	    CALL HAL$CLONE$1(1,R)
	  ELSE
	    CALL HAL$CLONE$2(9000,R)
	  ENDIF
	ENDIF
	...
	SUBROUTINE HAL(NA,RI)
	INTEGER NA !NA=0
	REAL RI(*)
	RI(1)=0.0
	END
	
	SUBROUTINE HAL$CLONE$1
	INTEGER NA !NA=1
	REAL RI(*)
	RI(1)=1.0
	END
	
	SUBROUTINE HAL$CLONE$2(NA,RI)
	INTEGER NA
	REAL RI(*)
	DO I=1,7000
	  RI(I)=1.0/I
	ENDDO
	END
	...</pre>

The Application Compiler can create clones at more than one level in a call tree. In the following example, the APC clones SUB1 so that it can propagate a constant to SUB2, which is also cloned.

Original code	Optimized code

PROGRAM MAIN
CALL SUB1(1000)
CALL SUB1(2000)
END
 	
SUBROUTINE SUB1(N)
INTEGER N
REAL R(2000)
CALL SUB2(N,R)
...
END
 	
SUBROUTINE SUB2(N,R)
INTEGER N
REAL R(*)
DO I=1,N
  R(I)=I+0.5
ENDDO
END
	PROGRAM MAIN
	CALL SUB1(1000)
	CALL SUB1$CLONE$1(2000)
	END
	
	SUBROUTINE SUB1(N)
	INTEGER N ! N=1000
	REAL R(2000)
	CALL SUB2(1000,R)
	...
	END
	
	SUBROUTINE SUB2(N,R)
	INTEGER N ! N=1000
	REAL R(*)
	DO I=1,1000
	  R(I)=I+0.5
	ENDDO
	END
	
	SUBROUTINE SUB1$CLONE$1(N)
	INTEGER N ! N=2000
	REAL R(2000)
	CALL SUB2$CLONE$1(2000,R)
	...
	END
	
	SUBROUTINE SUB2$CLONE$1(N,R)
	INTEGER N ! N=2000
	REAL R(*)
	DO I=1,2000
	  R(I)=I+0.5
	ENDDO
	END

A clone of a procedure can be reused; that is, it can be called again, either by the procedure which contains the call that created it or by another procedure. This reduces the number of clones the Application Compiler would otherwise need to create.

Automatic parallelization of loops with calls

In procedural compilers, loops that contain procedure calls (except, in some cases, library procedure calls) are not parallelizable. This is because the compiler cannot determine the side effects present in user-written called procedures, so it must assume that the procedures contain side effects that prevent parallelization.

The Application Compiler, however, has full access to side effect information on all procedures being compiled, including those that are called from loops. If the loop has no characteristics that would normally prevent parallelization other than procedure calls, the APC can analyze the called procedures and, if they are also free of parallelization-inhibiting characteristics, parallelize the loop. This capability exists on both C Series machines and SPP Series machines, but the biggest gains are realized on SPP Series machines, where the number of processors available to run the parallelized loop is typically larger.

Loops with calls parallelization vs. inlining and parallelizing

It is important to note that parallelizing loops with calls and automatic inlining are separate optimizations. Inlining procedures that are called from otherwise parallelizable loops can substantially increase parallelization, but all inlining is done before parallelization analysis. Calls are inlined according to a set of heuristics that are controllable by the user, as described in Chapter 6, "Directives". After inlining is done, any calls left inside otherwise parallelizable loops are analyzed, and if they do not contain any parallelization inhibitors, the loops can be parallelized. If you discover that a parallelizable loop containing a call is being inlined and suspect that performance may increase if the loop is parallelized with the call intact rather than with the procedure inlined, you can prevent inlining with the `NO_INLINE` directive, which is described in Chapter 6. If the call contains no parallelization inhibitors, this may allow the APC to parallelize it; however, it will not guarantee a performance increase.

Call eligibility

When examining loops containing calls to determine whether they are eligible for parallelization, it is important to remember that the ultimate goal is to get copies of the called procedure running on several processors simultaneously. In order to achieve this, the called procedure, like any entity in a parallel loop, must not contain any standard inhibitors of parallelization, such as loop

carried dependencies. Unfortunately, the information available to the APC on data dependencies is not as precise for call data as it is for loop data, and loop transformations are more difficult to perform when a call is involved. These factors increase the likelihood of parallelization inhibitors, and, from the compiler's point of view, severely complicate the task of parallelizing a loop containing a procedure call.

Private data

Keeping this in mind, you can usually determine whether a loop containing a call is parallelizable by the APC, or why a specific call failed to go parallel, by examining the procedure's data.

In order to parallelize a loop containing a call to a procedure, the procedure's arrays and variables must be *privatizable*. Private data cannot include variables that must maintain static values between calls to the procedure in question. The APC can privatize certain data under certain circumstances, as described in the "Data privatization" section, which follows.

Any array data passed to the procedure must also be disjoint. This means that an array section assigned during one call to the procedure must not overlap with any sections assigned or referenced during another call.

The following example shows a parallelizable call in which the array sections referenced by the procedure on each call are disjoint.

```
PROGRAM DRIVEPROD
INTEGER A(30,50), B(30,50), C(30,50)
...
DO J = 1, 50
  CALL PROD(A(1,J),B(1,J),C(1,J),30)
ENDDO
END

SUBROUTINE PROD(A, B, C, ROWS)
INTEGER A(*), B(*), C(*), ROWS
DO K = 1, ROWS
  C(K) = A(K) + B(K)
ENDDO
RETURN
END
```

In this example, the arrays `A`, `B` and `C` are passed to the procedure `PROD` one column per call in the `J` loop of the main program. The `ROWS` formal argument contains the total number of rows in each array (all three arrays are of the same shape), insuring that the entire column is processed on each pass. The columns are passed

into single-dimensional arrays in the subroutine to simplify processing. Since entire columns are passed on every call, the data is disjoint, and the Application Compiler can parallelize the `J` loop.

Remember, when a loop containing a call is running in parallel on several processors, each processor will be independently running a call to the procedure as part of its share of loop iterations. If any operation in the procedure call depends on a previous call, it is likely that the calls will not execute in the correct order on their respective processors and wrong answers will result. It is for this reason that calls contained in loops must pass such rigorous heuristics and/or be modified either manually or by the APC before the loops can be parallelized.

APC optimizations

Self-dependence

When examining a procedure to see if a loop containing a call to it is parallelizable, the APC first determines whether the procedure is self-dependent. Self-dependent procedures contain local static arrays and/or scalars that are defined in one call to the procedure and used in a later call. Typically, C-language self-dependent procedures make use of variables declared within the procedure to be of the `static` storage class. Fortran-language self-dependent procedures typically employ `SAVE` statements or rely on default static local storage on C Series machines. Calls to self-dependent procedures *cannot* be parallelized, and the Application Compiler cannot convert self-dependent procedures into non-self-dependent procedures. If a procedure is written in a self-dependent manner and you would like to parallelize a loop that calls it, the procedure must be rewritten in a non-self-dependent manner.

In order to rewrite a self-dependent procedure to be non-self-dependent, you must find the portion of the procedure that is causing the self-dependence and remove it. While this is seldom a trivial task, most procedures that contain self-dependences can be modified so that the self-dependent code is executed in the main program rather than the procedure, allowing the procedure to be parallelized if no other parallelization inhibitors are present. This of course assumes that there is enough code left in the procedure after self-dependence removal to not only warrant the procedure call, but to make parallelization of the loop containing the call more profitable than inlining the rewritten procedure.

The following example shows a self-dependent Fortran subroutine and its calling loop that have been rewritten to be non-self-dependent.

Fortran example: self-dependent	Fortran example: non-self-dependent

<pre>DO I = 1, N
  CALL FINDPOS(I, N, A, IA)
ENDDO
 	
...
 	
SUBROUTINE FINDPOS(IX, N, A, IA)
INTEGER IX, N, IA(*)
REAL A(*)
DATA K/0/ !IMPLIES SAVE FOR K
 	
NN = N - IX
XX = 1/REAL(IX*IX)
Y = (NN * COSD(REAL(IX))) + XX
   .
   .
   .
 	
IF(Y .GT. 0) THEN   !K CAUSES
  K = K+1   !SELF-DPNDNC
  A(K) = Y
  IA(K) = IX
ENDIF
 	
RETURN
 	
END</pre>
	<pre>DATA K/0/
	
	...
	
	DO I = 1, N
	  CALL FINDPOS(I, N, Y(I))
	ENDDO
	
	DO J = 1, N
	  IF(Y(J) .GT. 0) THEN
	    K = K+1
	    A(K) = Y(J)
	    IA(K) = J
	  ENDIF
	ENDDO
	
	...
	
	SUBROUTINE FINDPOS(IX, N, Y)
	INTEGER IX, N
	REAL Y
	
	NN = N-IX
	XX = 1/REAL(IX*IX)
	Y = (NN * COSD(REAL(IX))) +XX
	   .
	   .
	   .
	
	RETURN
	
	END</pre>

Note

This example assumes that `FINDPOS` contains adequate work to justify parallelizing the loop that calls it. If `FINDPOS` does not contain adequate work, the Application Compiler would likely inline it rather than parallelize the `I` loop.

The self-dependence in this example is caused by the variable `K`. The `DATA` statement initializes `K` to `0` and allocates it static storage at link time. `K` is then used as an array index to maintain the location of the positive values of `Y` in an array. In order for `FINDPOS` to work, it must maintain `K`'s value between calls. In order to eliminate this self-dependence, all operations that use `K` must be moved to the main program.

This is easily accomplished by storing each value of `Y` computed in `FINDPOS` in an array, and moving the block of code that locates

and stores the positive values into a loop in the calling procedure. In the non-self-dependent example, this is done by the calling procedure's J loop. The loop containing the call to FINDPOS can then be parallelized.

Similar self-dependence problems can exist in C code; however, because of C's frequent use of pointers, it is much more difficult for the Application Compiler to parallelize loops containing calls to procedures that manipulate arrays or any pass-by-reference data. Rewriting a C-language version of the previous Fortran example to be non-self-dependent would not necessarily allow the loop containing the call to be parallelized, because the dereference necessary to handle the single element of Y in the C version of FINDPOS would create a potential LCD.

When the existence of pointer dereferences in a C procedure is inhibiting parallelization of loops that call the procedure, the -alias array_args, -alias ptr_args, and -alias restrict_args C compiler options may help parallelize the loop. Refer to the C *User's Guide* for more information.

Apparent self-dependence

It is possible for a non-self-dependent procedure to appear to be self-dependent to the Application Compiler. An apparently self-dependent Fortran procedure is shown below.

```
SUBROUTINE NSD(STATE, A)
INTEGER STATE, A(*)
INTEGER AMB
IF(STATE .EQ. 0) THEN
  AMB = 1
ENDIF
IF(STATE .EQ. 1) THEN
  AMB = 2
ENDIF
DO I = 1, 1000
  A(I) = FUNC(I, AMB)
  ...
ENDDO
RETURN
END
```

If STATE always has the value 0 or 1, one of the IF blocks always executes and AMB is always assigned before it is used in the I loop; assuming this is the case, this procedure is non-self-dependent.

On C Series machines, AMB (like all local variables) is allocated static storage by default, and since the value of STATE is unknown at compile time, the compiler must assume that it may take a value

other than 0 or 1. The compiler therefore assumes that the procedure is self-dependent.

Specifying the `-re` option for NSD in the buildfile would force AMB into automatic (stack-based) storage; this would indicate to the APC that the procedure was in fact non-self-dependent and any loops that called it would be considered for parallelization.

Such apparent self-dependences which can be fixed by reentrant compilation are not a problem on SPP Series machines, where `-re` compilation is the default. Note, however, that actual self-dependences of this form will yield wrong answers if compiled on SPP Series machines at the default `-re` level. Such procedures must be explicitly compiled (on SPP Series only) using the `-nore` option, or rewritten to be non-self-dependent.

Procedure purity

If the procedure under consideration has been determined to be non-self-dependent, the Application Compiler can proceed to its next test, which is for procedure purity.

A procedure is considered to be *pure* if it doesn't depend upon or change the state of global variables. Global variables are typically stored in COMMON blocks in Fortran, or declared to be of the `extern` storage class in C procedures (and declared independently of all procedures that call them in the C source). Loops containing calls to pure procedures can be automatically parallelized by the Application Compiler. Unpure procedures cannot be executed in parallel, but can generally be automatically purified by the APC. To purify a procedure, the APC converts the global variables accessed by the procedure into arguments.

Automatic purification can be inhibited when Fortran COMMON block variables are aligned differently in different procedures that use them, as are the arrays A and B in the example below.

```
SUBROUTINE ALI1()
INTEGER A(100), B(100,100)
COMMON /BLK1/ A,B
...

SUBROUTINE ALI2()
INTEGER A(100,100), B(100)
COMMON /BLK1/ A, B
...
```

Purification is also inhibited when the procedure does I/O, or calls an unpurifiable or self-dependent procedure.

If any of these inhibitors are present, you must rewrite the procedure to eliminate them in order for purification to work.

The following example shows unpure and pure versions of the same procedures in both C and Fortran. While the APC won't rewrite your source code as shown in this example, after purification the internal representation of the procedure will function in a manner similar to the purified source shown, and the calling procedure's argument list will be appropriately modified.

C example: unpure procedure

```
void puree(n)
int n;
{
  extern int arr[];
  int i;
  for(i=0;i<n;i++) {
    arr[i] = ...
    ...
  }
}
```

C example: purified procedure

```
void puree(arr,n)
int arr[],n;
{
  int i;
  for(i=0;i<n;i++) {
    arr[i] = ...
    ...
  }
}
```

Fortran example: unpure procedure

```
SUBROUTINE PUREE(N)
INTEGER N
INTEGER IARR(100)
COMMON /BLK1/ IARR

DO I = 1, N
  IARR(I) = ...
  ...
ENDDO
...

RETURN

END
```

Fortran example: purified procedure

```
SUBROUTINE PUREE(IARR, N)
INTEGER IARR(100), N

DO I = 1, N
  IARR(I) = ...
  ...
ENDDO
...

RETURN

END
```

Procedure purification is an automatic interprocedural optimization. Provided none of the inhibitors mentioned above are present, no modification of source is necessary.

Data privatization

Once the Application Compiler purifies the procedure under consideration (or determines it to be pure), it can privatize all the procedure's local variables by compiling the procedure for reentrancy. In reentrant procedures, all local variables are dynamically allocated on the stack at each procedure call.

Reentrant compilation, achieved via the `-re` option to the procedural C Series compilers, is default behavior on all Convex SPP Series compilers.

Note

Compiling self-dependent procedures with `-re` can give wrong answers. If you know or suspect that your procedure is not reentrant, use the `-nore` option on that procedure to prevent reentrant compilation, or rewrite the procedure in a reentrant manner.

Default compilation on C Series compilers is non-reentrant, but if the APC can determine that it is safe to compile a procedure for reentrancy, it will do so automatically. However, the APC will choose reentrant procedures according to conservative heuristics to avoid using `-re` when it is unsafe; therefore, if you are compiling procedures which you know to be reentrant on a C Series machine, you may wish to specify `-re` for those procedures in the buildfile to insure reentrant compilation and increased opportunities for parallelization of loops with calls.

Related procedural compiler directives

The following procedural compiler directives can sometimes help and sometimes hinder the Application Compiler's automatic parallelization of loops with calls:

- `LOOP_PARALLEL` (SPP Series) or `FORCE_PARALLEL` (C Series)
- `PREFER_PARALLEL`
- `NO_PARALLEL`
- `THREAD_PRIVATE` (SPP Series only)
- `NODE_PRIVATE` (SPP Series only)
- `LOOP_PRIVATE`

The directives in this list that apply to C Series are described in detail in the *Fortran Optimization Guide* and *C Optimization Guide*. Those that apply to SPP Series are described in the *Exemplar Programming Guide*.

If the APC is missing parallelization opportunities or incorrectly parallelizing certain code, the directives listed above provide a means to manually correct the problem.

The following sections discuss situations in which these directives can either aid or hinder parallelization of loops with calls.

SPP Series only

APC optimizations

Manual data privatization

SPP Series machines support the `node_private` memory class, which is private to the processors on a given hypernode, and the `thread_private` memory class, which is private to a specific execution thread. These classes are assigned using compiler directives in Fortran and storage classes in C. They can be used to privatize data for the APC, but using them requires you to take complete control over parallelization of any code that accesses the private data. Proper use of these memory classes on SPP Series machines is therefore rather complicated and beyond the scope of this guide. It is, however, fully covered in the *Exemplar Programming Guide.*

It is less complicated to use the `LOOP_PRIVATE` directive to privatize data for the APC on SPP Series machines. Use of this directive, which is available on both SPP and C Series machines, is covered here.

Using `LOOP_PRIVATE` allows the APC to automatically parallelize nested loops in the most efficient manner for the target machine; you need not specify parallelization directives in addition to these directives unless your code appears unparallelizable to the APC.

Consider the following Fortran loop and subroutine.

```
...
INTEGER A(1000),C(1000),I,J
...
DO I = 1,N
  DO J = 1, M
    A(J) = ...
    ...
  ENDDO
  CALL LOOPK(A, M, C)
ENDDO
...

SUBROUTINE LOOPK(A, M, C)

INTEGER A(*), M, C(*)
DO K = M, 1
  C(K) = A(K) ...
  ...
ENDDO
...
RETURN
END
```

There are several opportunities for parallelism here, but each has its caveats.

The I loop is most preferable to parallelize because it is the outer loop. To parallelize a loop with a call, its data must be privatizable. A is first assigned and later used within the body of the I loop, so A (along with I, J, K, and C) is private in the context of that loop. You will note that A is actually assigned in a sub-loop (the J loop) of the I loop, and later used in another sub-loop (the K loop in subroutine LOOPK) of the I loop. As long as the I loop is chosen for parallelization, this does not present a problem; each I loop thread will have its own private copy of A, and the J and K loops will run scalar within each thread, assigning and using A in the right order.

Alternately, the I and/or J loops can be parallelized, but if A is privatized and the J loop is parallelized, C is likely to be assigned incorrect answers in the K loop, because the private A will not necessarily keep values assigned within the J loop.

You can insure that the Application Compiler parallelizes the I loop, which is most profitable, by using privatization directives. As an added benefit, specifying which variables to privatize and which loops to parallelize will save the APC substantial analysis time, especially on more complicated loops. The easiest solution is to use a LOOP_PRIVATE directive, as shown below.

```
      ...
      INTEGER A(1000),C(1000)
      ...
C$DIR LOOP_PRIVATE(A)
      DO I = 1,N
        DO J = 1, M
          A(J) = ...
          ...
        ENDDO
        CALL LOOPK(A, M, C)
      ENDDO
      ...
```

Here LOOP_PRIVATE indicates that A is private to the I loop only. Given this information, the APC will parallelize the I loop.

LOOP_PRIVATE can also help you solve a problem similar to self-dependence that exists at the loop level. Consider the following loop and subroutine.

```
...
DO I = 1, 1000
  CALL INIT(I, IVALUE)
  A(I) = IVALUE*B(I)
  ...
ENDDO
...
```

```
SUBROUTINE INIT(I, IVALUE)
INTEGER I, IVALUE
IF((I .GT. 0) .AND. (I .LE. 500)) THEN
  IVALUE = 0
ENDIF
IF((I .GT. 500) .AND. (I .LE. 1000)) THEN
  IVALUE = 1
ENDIF
END
```

If I is always between 0 and 1000, IVALUE will always be assigned in INIT, no self-dependence problem will exist, and it will be safe to parallelize the I loop. If you know this to be the case, you can use a LOOP_PRIVATE directive in the calling DO loop as shown below:

```
...
C$DIR LOOP_PRIVATE(IVALUE)
      DO I = 1, 1000
        CALL INIT(I, IVALUE)
        A(I) = IVALUE*B(I)
        ...
      ENDDO
      ...
```

This tells the Application Compiler that I and IVALUE are both always assigned before they are used on each iteration of the following DO loop, indicating that, provided no other parallelization inhibitors exist, the loop is safe to parallelize.

Refer to the *Exemplar Programming Guide* for more general information on using the PRIVATE and PARALLEL directives on SPP Series machines. Refer to the *Fortran Optimization Guide (C Series)* or the *C Optimization Guide (C Series)* for more information on using these directives on C Series machines.

Forcing parallelization

The LOOP_PARALLEL (SPP Series) and FORCE_PARALLEL (C Series) directives allow you to manually force parallelization of loops with calls that you are absolutely certain are parallelizable, but that are not automatically parallelized by the APC.

If you choose to use a directive to manually parallelize a loop that the APC will not parallelize, you must

- insure that the loop and any procedures it calls, directly or indirectly, contain no inhibitors of parallelization.
- insure that any called procedures are compiled for reentrancy (which is the default on SPP Series machines, but requires the -re option on C Series machines).

- on SPP Series machines, manually privatize any data used within the loop to be parallelized. On C Series machines, privatization is automatic.
- on SPP Series machines, insure that inlining will not be performed on a procedure call occurring within a LOOP_PARALLEL loop. Since LOOP_PARALLEL prevents automatic privatization of loop data, inlining can result in wrong answers if it is performed rather than calling the procedure in parallel. Refer to the "Forcing parallelization on SPP Series" section for more information.

Note

When you specify either manual parallelization directive, the APC assumes that you have taken care of these issues, and ceases further analysis of the loop. Wrong answers will result if you fail to manually handle any of the above conditions.

The LOOP_PRIVATE directive is the most automated privatization directive; using it may even eliminate the need for manual parallelization. The merits of these directives as opposed to THREAD_PRIVATE and NODE_PRIVATE are discussed in the "Manual data privatization" section.

Assume that a loop containing the apparently self-dependent procedure NSD discussed in the "Self-dependence" section fails to parallelize on an SPP Series machine, where default compilation is for reentrancy. If you are sure the procedure is not self-dependent, and that the loop contains no inhibitors of parallelization, you can force parallelization by both manually privatizing STATE by specifying a LOOP_PRIVATE directive on the calling loop along with a LOOP_PARALLEL directive and a NO_INLINE directive (to insure that the procedure is called in parallel rather than inlined, thus avoiding privatization problems with compiler-generated variables), as shown below.

```
      ...
C$DIR LOOP_PRIVATE(STATE)
C$DIR LOOP_PARALLEL
      DO J = 1, N
        STATE = MOD(J,2)
C$DIR   NO_INLINE
        CALL NSD(STATE, A)
        ...
      ENDDO
      ...
```

Enhanced error checking

Information available to a compiler determines any error checking the compiler can do. When procedures are compiled, certain errors are unavoidably overlooked because the compiler cannot see any code outside the scope of the procedure being compiled. The Application Compiler, however, has access to the information (obtained during analysis and synthesis) on other procedures. With this information, the APC finds more errors, resulting in more reliable and robust code when the errors are corrected.

The errors that procedural compilers miss that the Application Compiler catches include

- Calls that pass too few or too many arguments
- Calls that pass arguments with an incompatible type
- Calls that pass array arguments to scalars or scalars to arrays
- Global scalar or array variables that are not assigned an initial value
- Certain array accesses that go beyond the bounds of the array
- Procedures that are defined but not called
- Global variables with different declarations in different source files
- Aliasing errors
- Errors in COMMON-block layout
- ANSI 77 and ANSI 90 standards checking for Fortran

Procedural Fortran compilers cannot check the number and type of arguments passed to a procedure, for example. As a result, undetected typing errors can lead to runtime problems. Because the Application Compiler has information on the entire program, it can detect when the arguments passed do not agree with what the procedure expects.

If a call passes an incorrect number of arguments to a procedure, the Application Compiler issues a warning message.

One of the most frequent errors to afflict programs is writing or accessing a value beyond the bounds of an array. Because the program is reading from or writing to memory space that belongs to another variable or code, the results are unpredictable. Usually this problem occurs in a loop. Accesses that fall outside the bounds of an array are often impossible for ordinary compilers to detect because loop start and stop values may not be fixed.

In the following example, the loop in subroutine OVER has start and stop values that are passed in from another procedure. If these values are propagated constants, the APC can determine whether the loop uses memory outside the bounds of the array.

```
 ...
CALL OVER(AR, 1, 100)
 ...
SUBROUTINE OVER(AR, N, M)
REAL AR(200)
INTEGER N, M
DO I=N, M
  AR(I)=N/M
ENDDO
END
```

The Application Compiler also checks for many types of ANSI 77 and ANSI 90 conformance errors. ANSI standards checking involves both procedural and interprocedural analysis and provides functionality similar to what the lint facility provides for C. Because of the large number of error messages that older Fortran programs may generate, the APC does not perform these checks by default. You must specifically request them using command line options described in Chapter 3, "Using the Application Compiler."

Alias and pointer tracking

Aliasing is the use of multiple names to refer to the same object. The object is a memory location, and the name is a variable name. Aliasing poses problems for optimizing compilers. If detected, aliasing can force a compiler to give up trying to optimize a part of your code. If undetected, aliasing can lead to runtime errors. In C, aliasing problems are usually caused by the use of pointers or address operators. Although the Fortran language lacks C-like pointers, aliasing problems occur in Fortran programs due to argument passing and EQUIVALENCE statements. The Application Compiler identifies many of the aliasing problems likely to occur in your code.

Figure 9 shows a simple loop that has no aliases. The calculations performed on each pair of array elements are independent of one another: one iteration does not overlap or feed into the next. The loop is free of recurrences and its data can be safely vectorized on C Series machines.

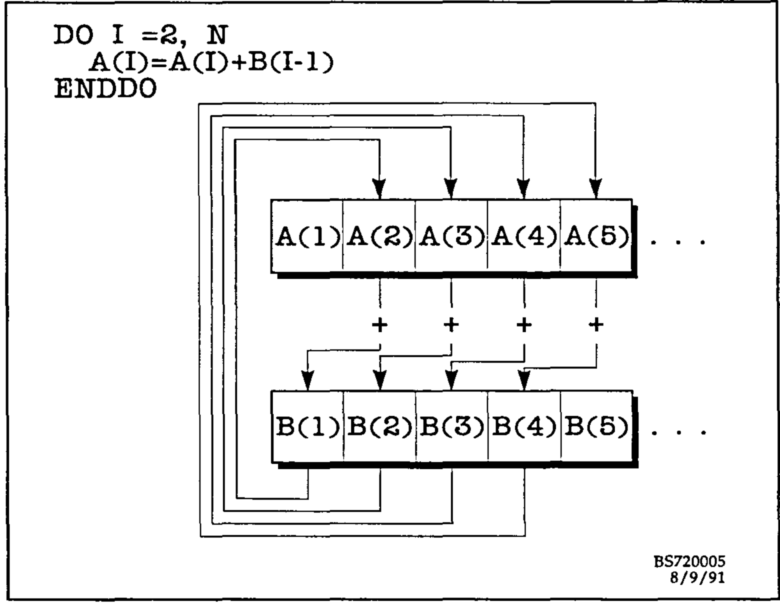


Figure 9 Loop without alias

Now look at Figure 10. This figure shows the same loop, with one difference: `A` and `B` are two different names, or aliases, for the same memory location.

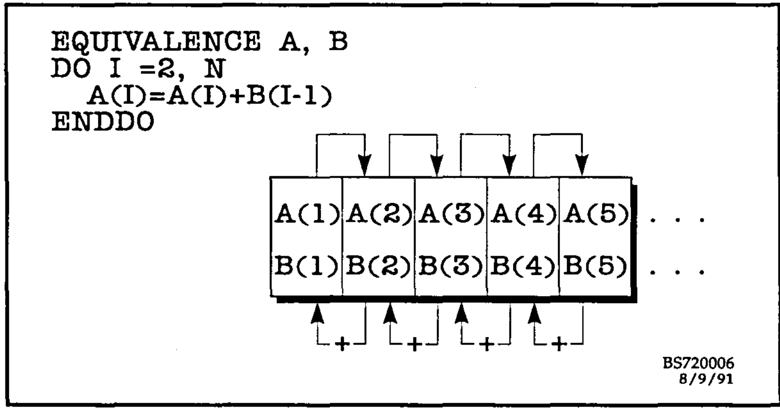


Figure 10 Loop with alias

As the arrows show, the calculation performed on one pair of array elements feeds into the next calculation. The iterations are now dependent on one another and a recurrence exists. As long as the loop is run in a serial, or scalar, manner, this poses no problem. But if the loop must be strip mined for vectorization on C Series machines, incorrect output results. Similar aliasing problems can arise on SPP Series machines in nested loops that are candidates for data localization.

Aliasing problems occur most often in C programs that use pointers. Pointers can be set to point anywhere in memory, so the compiler must have some algorithm to determine whether the locations pointed to by two pointers are aliases. The algorithm used by older Convex C compilers (and used by the current C compiler in backward-compatible mode) assumes that all pointers are subscripts into a single giant array that encompasses all of memory. All pointers are therefore considered to be potential aliases for one another.

When invoked in ANSI mode, the compiler uses a somewhat stricter alias detection algorithm that assumes a separate memory-wide array for pointers of each base type. This algorithm finds fewer potential aliases, but it is only valid if the program adheres to ANSI restrictions on type compatibility. Many older programs are not ANSI compliant, and converting an existing program to conform to ANSI specifications can be a formidable task.

Pointer tracking

Note

The pointer tracking algorithm is experimental in nature and may occasionally perform unsafe optimizations. For this reason, pointer tracking is not performed by default. To enable pointer tracking, specify the `-enable pointer_track` option on the `apc` command line.

When the `-enable pointer_track` option is specified, the Application Compiler uses a much more powerful alias detection algorithm called *pointer tracking*. Through interprocedural analysis, the APC can track the memory locations to which each pointer is set. As a result, the APC reduces the number of potential aliases.

Consider the following C code example:

```
track() {
int *p, *q, *r;
int i;
p = malloc(100 * sizeof(int));
q = malloc(100 * sizeof(int));
r = malloc(100 * sizeof(int));
...
for(i= 0; i<100; i++) {
*p++ = *q++ + *r++;
}
}
```

To most procedural compilers, `malloc()` is just another function with unknown side effects that might create aliases between `p`, `q`, and `r`. This possible alias would, on C Series machines, prevent vectorization of the `for` loop. Through interprocedural analysis, the Application Compiler can determine that `malloc()` creates no aliases and that the loop is safe to vectorize or parallelize, depending on the hardware platform and optimization level being used.

Pointer tracking also resolves problems with aliasing between pointers and global variables, static variables, formal parameters, and variables that have their address taken using the `&` (address-of) operator. By resolving these aliasing problems, the Application Compiler can greatly increase the number of loops that can be data-localized in a C program.

Pointer tracking and enhanced error checking provided by the Application Compiler also help to eliminate the problem of hidden aliasing, which occurs primarily in Fortran. Hidden aliases remain undetected by the procedural compiler. Hidden aliases can cause the Fortran compiler to incorrectly optimize code. Usually, these aliases result from code that does not conform to the language standards that the compiler implements.

The following code, shown in Fortran and C, has a hidden-alias problem. The procedure `CONFUSED` expects to receive three separate arrays. Instead, it is passed the same array (`A`) twice. The ANSI standard for Fortran declares that these dummy arguments must not be the same. (For a complete description of the ANSI Standard statement as it applies to aliases, see Chapter 9 of the *Fortran Optimization Guide (C Series).)* In C, array parameters need not be separate. If you use the `-alias array_args` option, however, the compiler assumes that they are separate. The procedural compilers do not detect aliasing between the arrays `X` and `Z`. (The C example is compiled with `-alias array_args.`) On C Series machines, the loop in `CONFUSED` is vectorized even though a recurrence exists, and a wrong answer results.

Fortran code	C code

PROGRAM MAIN
 	
REAL A(500),B(500)
CALL CONFUSED (A,B,A,500)
END
	main()
	{
	float A[500], b[500];
	confused(A, b, A, 500);
	}

SUBROUTINE CONFUSED (X,Y,Z,N)
INTEGER N
REAL X(*),Y(*),Z(*)
DO I=2,N
  Z(I)=Y(I)+X(I-1)
ENDDO
RETURN
END
	confused(x, y, z, n)
	int n;
	float x[], y[], z[];
	{
	 int i;
	 for (i=1; i < n; ++i)
	   z[i]=y[i]+x[i-1];
	}

Hidden aliasing can also result from the use of Fortran COMMON blocks. The following code is a Fortran program with an alias.

```
PROGRAM MAIN
INTEGER N
COMMON N

N=789
CALL CONFUSED(N)
END

SUBROUTINE CONFUSED(N)
DO I=1,2
  N2=3*(N+1)
  CALL CALC(N2)
  WRITE(*,*) 'Iteration:', I, ',   N= ', N
  WRITE(*,*) 'Iteration:', I, ',   N2= ', N2
ENDDO
RETURN
END

SUBROUTINE CALC(N)
INTEGER K, N
COMMON K
K=N+1
END
```

N appears to be invariant within the I loop, but because it appears in a COMMON block in MAIN, it is an alias for K, which appears in a COMMON block in CALC. N is altered on every call to CALC because K is altered on every call to CALC. The procedural compiler does

not recognize this and incorrectly optimizes the loop by moving the line that references N and assignments to N2 out of the loop.

When invoked with the `-show aliases` option, the Application Compiler detects and reports aliasing between formal parameters (dummy arguments) and between dummy arguments and COMMON block variables. The `-show pointers` option can also help you find aliases. Both of these options are described in Chapter 3, "Using the Application Compiler".

For more information on aliasing, consult the *Fortran or C Optimization Guide (C Series).*

APC optimizations

Link optimization

The Convex Application Compiler optimizes the order in which object files are linked to produce the most efficient possible executable. Link optimization reduces the number of page faults and instruction-cache misses.

Link optimization places procedures that call one another close together in the finished executable. The APC estimates the probability that two procedures will be on the execution stack at the same time and sorts procedures according to these probabilities.

By default, the compiler calculates these probabilities based on estimated call frequencies. If profiling data is available, it can use the actual, measured call frequencies instead. When profiling data is available, the `-prof`, `-gprof` (C Series only) and `-pdf` (both platforms) options allow the Application Compiler to optimize linking based on the execution time, rather than the frequency, of procedure calls. Refer to Chapter 3, "Using the APC," for more information.

Memory bank/ cache line optimizations

Memory-bank conflicts occur on C Series machines when a nonmajor dimension of an array (a dimension other than the rightmost dimension in Fortran or leftmost dimension in C) is an integral multiple of the number of memory banks or *interleave.* They occur on SPP Series machines when the nonmajor dimension is not an integral multiple of the number of elements of the array type that fit in a processor cache line.

The Application Compiler can optimize many of these arrays to reduce bank conflicts. If certain conditions are met, the C Series compiler extends the troublesome dimensions to `1+(`*interleave*`*(1+`*ol*`/`*interleave*`))` where *ol* is the original length of the dimension and the division is integer division.

On SPP Series machines, the Application Compiler optimizes array storage (if necessary) by extending nonmajor dimensions so that they are integrally divisible by the number of elements of the given array type that fit in a cache line.

Array storage optimizations are controllable using the `-extend_dim multiple` and `-extend_dim all` options on C Series machines. These options are not available on the SPP Series machines. Array storage can be disabled on both platforms through use of the `-extend_dim none` option. All `-extend_dim` options are described in detail in Chapter 3.

The Application Compiler cannot extend the dimension of the following arrays:

- Arrays that are in a Fortran equivalence with other arrays of different dimension or element size
- Arrays that are passed as arguments with reshaping (a change in dimension from one procedure to another)
- Arrays that are input or output as a whole (arrays that are input or output by elements are okay).
- Arrays that are accessed with unsafe subscripts

Many subscript accesses do not provide sufficient information for the Application Compiler to verify their safety. If you are certain that your program does not over- or undersubscript arrays, you can tell the compiler using the `-assert subscripts_ok` option. For more information on bank conflicts on C Series machines, see Chapter 6 of the *Fortran* or *C Optimization Guide (C Series)*.

Using the Application Compiler

3

This chapter shows you how to compile a program using the Application Compiler. It describes how to use the apc command and its associated command line options to compile and optimize a program.

Optimizing with the Application Compiler

Because it automatically optimizes a wider range of programming constructs, the Application Compiler reduces the need for manual intervention during the optimization process. The mechanics of optimizing a program are also simplified. Inlining of procedures, for example, is now completely automatic, and the need to use optimization directives has been reduced.

Keep in mind that, while the Convex Application Compiler is a breakthrough in speed for interprocedural optimization, it is slower than the conventional Convex C and Fortran compilers. You may want to use the conventional compiler for some stages of your program's development to save time. If you are porting a complete program, use the Application Compiler for the first (debug) version; the additional error checking that the Application Compiler offers more than makes up for the additional compile time. You can use the conventional compilers to compile the intermediate versions, then switch back to the Application Compiler to compile at optimization level -O2 or higher. A recommended strategy for optimizing programs is presented in Chapter 7, "Optimization strategy".

There are two steps in using the Application Compiler:

1. Create a buildfile.
2. Invoke apc.

The buildfile is a text file that lists files, libraries, and compilation options for the apc command. The apc command depends on

buildfiles the same way the make utility depends on makefiles. The buildfile tells apc what to compile and what compiler options to use. Compiling a program with the apc command is sometimes referred to as *building* the program.

A makefile must specify the compilation dependencies, or relationships between files that require compiling in a specific order. Because apc scans and analyzes your program source files, it can determine these dependencies automatically. A buildfile is therefore simpler than a makefile. Usually, a buildfile need specify nothing more than a directory or list of files to compile and a list of options to use in compiling them. These are the same compilation options that appear on the Convex Fortran or C compiler command line. You can create a buildfile using any ASCII text editor; Chapter 4, "Creating a buildfile", gives complete instructions on how to write a buildfile.

To invoke the Application Compiler, enter the command apc, followed by appropriate options, at the shell prompt. Compiling and optimizing are automatic and require no human intervention from that point on. While compiling an application, the Application Compiler gives the following messages and reports:

1. Language-analysis messages
2. Interprocedural error messages
3. Optimization messages
4. Optimization report
5. InterProcedural-Optimization (IPO) report
6. Status messages, which show the order in which procedures compile

Inlining and cloning can change the order in which procedures compile.

The language-analysis messages are the same as those the conventional compiler generates for the language you are using. The optimization report is substantially the same, with some added features for cloned and inlined procedures. Consult the *Fortran User's Guide* or *C User's Guide* for a detailed explanation of the compiler messages for those respective languages or the appropriate optimization guide for how to read the optimization report. The "Optimization report" section of this chapter explains the multiple optimization reports generated for inlined and cloned procedures.

Interprocedural error messages provide warning information about errors that are undetectable with conventional compilers. (See Appendix C, "Error and warning messages", for a complete explanation of all warning and error messages.) Any analysis data tables that you request using the -show option also appear with

these messages. Optimization messages and the optimization report are identical to those produced by the procedural compilers. The IPO report is analogous to the optimization report, but includes only interprocedural optimizations.

The IPO report consists of two tables. The first table shows the interprocedural optimizations performed—constants propagated, calls inlined, and clones created—for each procedure. The second table shows the number of errors and warnings found for each procedure.

You can view the interprocedural messages and reports later using the `pdbview` utility. For information on the `pdbview` utility, which comes with the Convex Application Compiler, refer to Appendix B, "The PDB Viewer interface".

You can control whether these reports and messages are displayed. To turn off the language-analysis and optimization messages, use the `-nw` compiler option in your buildfile; to turn off the conventional optimization report, use the `-or none` compiler option. You can suppress most interprocedural messages by using the `-s` (silent) option on the `apc` command line. Other analyzer messages appear only if you specifically request them using the `-check` or `-show` options to `apc`. The `-nrep` option to `apc` turns off the IPO report.

The `-check`, `-show`, and `-nrep` options only affect the display of reports at compile time. The Application Compiler always creates this information and stores it in the program database, so you can examine it later with the Convex PDB Viewer.

Using the APC

Invoking apc

The `apc` command is the user interface to the Application Compiler. `apc` is invoked from the shell prompt. An `apc` command, with a few of its options, looks like this:

```
% apc -o pname -check types -inline low
```

The following sections categorize and describe the `apc` options.

apc control options

The options listed in this section control the `apc` program.

`-f`

The `-f`, or filename, option specifies the name of a buildfile. If you do not use this option, `apc` looks for a file named `buildfile`. If it fails to find that, it looks for a file named `Buildfile`. If it does not find either, `apc` cannot continue; it

aborts with the error message, `Error: cannot find 'buildfile' or 'Buildfile'`.

`-library`

This option tells `apc` to create an APC library instead of an executable. For full information on APC libraries, see Chapter 5.

`-o`

The `-o`, or output-name, option tells the Application Compiler to give the executable a specified name. To name the executable `mydb`, for example, use the following option:

```
% apc -o mydb
```

If you do not use this option, the Application Compiler names the output program after the current directory. If the current directory is `/mnt/me/work/cog`, for example, the Application Compiler creates an executable named `cog`.

`-vn`

The `-vn`, or version number, option causes `apc` to print out the version number of the Application Compiler you are using. This version number appears as a header before any other messages. See Figure 11.

```
% apc -vn

CONVEX APC V2.0.0
```

Figure 11 `-vn` option output

Compilation options

The options listed in this section control program compilation.

`-n`

The `-n`, or no-recompile, option tells `apc` not to recompile any files. Instead of compiling, `apc` generates messages that show which files need to be recompiled, as shown below:

```
OUT OF DATE SOURCE FILES:

-------------------------

ptr.c
```

-no *arg*

-no options disable certain Application Compiler features. This can be useful for speeding up compilations when you know a certain feature is not useful for a certain program, or for disabling features that may be causing compilation failures or wrong answers. All -no options are listed in Table 1.

Table 1 -no arguments

Option	Disables
-no array_anal	Array analysis
-no constant_prop	Constant propagation
-no scalar_anal	Scalar analysis
-no storage_opt	Storage optimization
-no ipo	Interprocedural optimizations
-no pure_anal	Procedure purification analysis

Note

Disabling one feature may also disable other features which depend on it.

-permit dynamic_binding

The -permit dynamic_binding option tells the Application Compiler to permit the inlining of Fortran procedures that require dynamic binding of array arguments. Dynamic binding is required to inline calls where the set of dimensions of an array being passed does not match the dimensions of the dummy argument it is passed to. Inlining of calls that require dynamic binding sometimes slows down code, so the Application Compiler does not inline these calls by default. Chapter 8, "Fortran hints", discusses dynamic binding in more detail.

-permit unresolved

The Application Compiler performs library resolution during part 1 of the synthesis phase. By default, if the APC encounters unresolved symbols during this phase, it issues an error message and terminates compilation. The -permit unresolved option causes the APC to issue a warning rather than an error if it encounters unresolved symbols during part 1 of the synthesis phase, and to continue with compilation until the link phase, where it will terminate. Libraries containing the unresolved symbols can then be linked

manually. However, doing so will impair interprocedural optimization to the same extent as using unannotated libraries.

`-time`

The `-time` option tells the Application Compiler to display a summary of the compilation time. Figure 12 shows a time summary.

```
Time (hr:min:sec.micro)  wall             user             system
------------------------------------------------------------------
Summary phase ...........0:07.845140      0:01.146127      0:06.006555
Synthesis phase (pass 1) 0:01.394301      0:00.039192      0:00.748977
Analysis phase ..........0:10.452956      0:00.883812      0:08.830694
Synthesis phase (pass 2) 0:01.056853      0:00.174980      0:00.758183
Compilation phase .......0:16.703625      0:05.359812      0:10.088493
Link phase ..............0:05.282128      0:01.559707      0:01.595821
Total build.. ...........0:43.818562      0:09.212236      0:28.742893
```

Figure 12 Time summary generated by `-time` option

Error control options

`-i`

By default, the Application Compiler stops compiling when fatal errors are found in one source file. The `-i`, or ignore failures, option tells the Application Compiler to ignore the failure and continue trying to compile other source files.

`-check` *arg*

The `-check` option specifies additional error-checking messages for the Application Compiler to print. Possible values for *arg* are shown in Table 2.

Table 2 `-check` option arguments

Argument	Checks
arrays	Over- or undersubscripted arrays
common	Common-block size and layout
init	Unitialized global and COMMON variables
types	Type errors
all	All of the above
only	All of the above; don't generate code

Each `-check` option is described in detail below.

`-check arrays`

Many runtime errors result from over- or undersubscripting an array. When you specify the `-check arrays` option, the Application Compiler looks for uses and assignments where over- or undersubscripting may occur, and generates error messages:

```
Warning: Assigns to array a may have
subscripts greater than upper bound in main

Warning: Uses of array n may have
subscripts greater than upper bound in main
```

`-check common`

The `-check common` option tells the Application Compiler to check for conflicts or inconsistencies in the declaration of Fortran COMMON blocks. Standard Fortran allows COMMON blocks to be declared with different sizes in different subprograms. In practice, however, mismatched COMMON blocks often result from typographical errors. Consider, for example, the following code. In subroutine SUB1, the programmer has mistakenly left out one of the two variables that should be in COMMON-**block** FOO.

```
PROGRAM MAIN
COMMON /FOO/A,B
REAL AR1(1000)

CALL INIT(A,B)
CALL SUB1(AR1,1000)
END
```

Using the APC

```
SUBROUTINE SUB1(AR1,N)
REAL AR1(*)
COMMON /FOO/A

DO I=1,N
  AR1(I)=(I+A)/B
ENDDO
END
```

If you compile this code with the conventional compiler, or the Application Compiler without `-check common`, you get the following message:

```
fc: Warning on line 17.23 of ./foo.f: 'B'
used before being assigned.
```

The compiler assumes that the symbol `B` in `SUB1` refers to an implicitly declared variable that has not been initialized. If you compile this same program using the Application Compiler's `-check common` option, you also get the following message:

```
Advisory: Common variable /FOO/A declared
with different types in procedure MAIN in
./foo.f and procedure SUB1 in ./foo.f.

Advisory: Common variable /FOO/B declared
with different types in procedure MAIN in
./foo.f and procedure SUB1 in ./foo.f.
```

This helps identify the cause of the problem more precisely. This option also detects type conflicts among `COMMON`-block variables. Consider, for example, the following code:

```
PROGRAM MAIN
COMMON /FOO/A,B
REAL AR1(1000),A,B
CALL INIT(A,B)
CALL SUB1(AR1,1000)
END

SUBROUTINE SUB1(AR1,N)
REAL AR1(*)
COMMON /FOO/A,B
INTEGER A,B
DO I=1,N
   AR1(I)=(I+A)/B
ENDDO
END
```

The `COMMON`-block variables A and B are declared as type `REAL` in `MAIN` and type `INTEGER` in `SUB1`. If you use `-check common`, the Application Compiler detects this conflict and issues the following message:

```
Advisory: Members of common block FOO
declared differently in procedure MAIN in
./foo.f and procedure SUB1 in ./foo.f
```

The `-check common` option also identifies COMMON blocks that have name conflicts with global variables. This happens often in mixed-language programs. Suppose, for example, you have a `main.f` file on a C Series machine that looks like this:

```
PROGRAM MAIN
COMMON /BLOCK/ A
END
```

In the same program, you have a C file that contains the following declaration:

```
extern int _block_;
```

The C Series compiler's internal symbol-table name for the global external variable `_block_` exactly matches the symbol table name for BLOCK. If not detected, this unintentional duplication could result in a logic error. The `-check common` option reports this:

```
Advisory: Common block BLOCK(__block_) and
global variable _block_ in foo.c refer to the
same memory location
```

`-check init`

When you specify `-check init`, the Application Compiler looks for COMMON or global variables that are used in a procedure without being initialized anywhere in the program.

Conventional compilers generally cannot tell whether a COMMON or global variable has been initialized in another procedure and so are unable to provide this type of error checking.

Consider the following program, which contains an error. The code neglects to initialize the COMMON array A to its proper values, leading to a possible divide by zero in SUB1.

```
PROGRAM MAIN
REAL B(1000)
COMMON /BLOCK/ A(1000)

CALL SUB1(B)

END
```

Using the APC

```
SUBROUTINE SUB1(B)
REAL B(1000)
COMMON /BLOCK/A(1000)

DO I=1,1000
 B(I)=1/A(I)
ENDDO

END
```

If you compile this program with `-check init`, the Application Compiler generates the following warnings:

```
Warning: MAIN uses A, which is neither
assigned nor initialized
Warning: SUB1 uses A, which is neither
assigned nor initialized
```

Because `SUB1` is called by `MAIN`, the Application Compiler generates a warning for `MAIN` as well as `SUB1`, even though `MAIN` does not directly use `A`. The Application Compiler reports an initialization error for a procedure whenever the procedure or one of its descendants has an initialization problem. Initialization errors also appear in the last column of the IPO report, as shown in Figure 13.

```
Errors Detected
---------------
                        Wrong  Mis-              Scalar
            Mis-        Number Matched           Passed            Variables
            Matched     Of     Return  Invalid   To      Invalid   Not
Procedure   Args        Args   Type    Aliases   Array   Subscript Initialized
------------ ------- ------ ------- ------- ------ --------- ----------
MAIN                                                                   1
SUB1                                                                   1
Totals                                                                 2
------------ ------- ------ ------- ------- ------ --------- ----------
build -check types   types  types   aliases arrays  arrays    init
```

Figure 13 IPO report with initialization errors

Note also that the -check init option depends on earlier array analysis. This does not mean that you must use -check arrays if you use -check init. It does mean, however, that -check init can miss some uninitialized variables that are not reported by array analysis. Consider, for example, the following code:

```
PROGRAM MAIN
COMMON /A/ B,C(10)
P=B
Q=C(5)
END
```

Even though array C is in a COMMON block, the Application Compiler knows that it is not used interprocedurally. It does not perform array analysis on C, and so initialization analysis misses that array.

Initialization analysis for scalars does not depend on array analysis. If you have an uninitialized scalar that is in COMMON, -check init can detect any use of that scalar whether it is used interprocedurally or not. It can detect the use of uninitialized COMMON-block scalars as well as arrays.

-check types

The -check *types* option tells the Application Compiler to display warning messages for procedure calls with type problems. The example below shows a warning message generated by -check types. If you need more complete information on a type mismatch, use the -show types option discussed later in this chapter.

```
Warning: Argument number 1 of DFUN has
inconsistent type in ./t.f on line 4 and
./t.f on line 8
```

-max_errors

This option allows you to increase the maximum number of errors reported by the Application Compiler. The Application Compiler has a limit on the number of errors it can report at various phases in the compilation process. On rare occasions, a program can exceed this limit.

If your compilation terminates with a "too many errors" message, use this option to increase the limit. By default, the Application Compiler terminates if 100 errors are detected in one phase. If your program reports too many errors, you can increase this by specifying a new limit, greater than 100, with -max_errors:

```
% apc -max_errors 200
```

Message control options

The options listed in this section allow you to control messages generated by the Application Compiler.

`-nrep`

This option suppresses printing of the interprocedural optimization (IPO) report.

`-s`

The `-s`, or silent, option prevents the Application Compiler from printing a message indicating that it is executing each phase as it begins the phase. Note that `-s` and `-v` are mutually exclusive. Figure 14 shows the messages that `apc` typically generates when compiling a file without `-s`. The optimization and IPO reports are abbreviated for clarity.

```
% apc
Computing Dependencies
Summary phase
Synthesis phase (pass 1)
Analysis phase
Synthesis phase (pass 2)
Compilation phase

<Opt Report for MX1>
<Opt Report for XM1>
<Opt Report for SB1>

Compilation: Warning on line 6.27 of
 ./sec.f: 'T' used before being assigned.
Compilation: Warning on line 10.3 of
 ./sec.f: 'T' used before being assigned.

<Opt Report for MAIN>
<Opt Report for MAIN.MGEN (Inline copy
 of MGEN in MAIN)>
<Opt Report for MGEN>
<IPO Report>
```

Figure 14 APC messages without `-s`

When the `-s` flag is used, `apc` compiles the same program without generating any phase messages, as shown in Figure 15.

```
% apc -s
<Opt Report for MX1>
<Opt Report for XM1>
<Opt Report for SB1>
Compilation: Warning on line 6.27 of
 ./sec.f: 'T' used before being assigned.
Compilation: Warning on line 10.3 of
 ./sec.f: 'T' used before being assigned.
<Opt Report for MAIN>
<Opt Report for MAIN.MGEN (Inline copy
 of MGEN in MAIN)>
<Opt Report for MGEN>
```

Figure 15 APC messages with -s option

-show *info*

The -show option tells the Application Compiler to display interprocedural analysis information to the screen. *info* can take one of the following values:

- aliases
- arrays
- calls
- clones
- constants
- inline
- pointers
- renames
- resizes
- scalars
- state
- symbols
- outlines
- types
- all

You can specify one or more of these -show options on the apc command line. These options produce additional messages or data tables that display detailed information from the analysis phase. Each is described in detail below.

Using the APC

-show aliases

The -show aliases option causes the Application Compiler to generate warnings for questionable aliases. For example, the following Fortran program passes the same actual argument to three dummy arguments, creating a set of hidden aliases. The ANSI Fortran standard prohibits this practice, although it is not detected by most Fortran compilers.

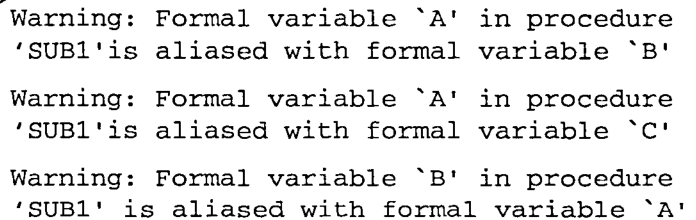
```
PROGRAM MAIN
REAL A(1000)

CALL SUB1(A,A,A)
END

SUBROUTINE SUB1(A,B,C)
REAL A(1000), B(1000), C(1000)

DO I=100,900
   A(I)= B(I-1)+C(I-1)
ENDDO
END
```

When you compile this program with -show aliases, the Application Compiler generates the warnings shown in Figure 16.



```
Warning: Formal variable `A' in procedure
'SUB1'is aliased with formal variable `B'

Warning: Formal variable `A' in procedure
'SUB1'is aliased with formal variable `C'

Warning: Formal variable `B' in procedure
'SUB1' is aliased with formal variable `A'
```

Figure 16 Warnings generated by -show aliases

-show arrays

Invoking apc with the -show arrays option causes the Application Compiler to display a table for arrays, like the one shown in Figure 17.

```
            Array Side Effects of Procedure MAIN (inter-procedural)
VARIABLE         USE                                ASG
--------         ---                                ---
A                (*,*)                              (*,*)
B                (*)                                (*)
IPVT             (*)                                (*)
X                (*)                                (*)
Z                (1,100)#

            Array Side Effects of Procedure SUB1 (inter-procedural)
VARIABLE         USE                                ASG
--------         ---                                ---
M               (1:N2,1:N1)
X               (1:1,1:N2)
Y               (1:1,1:N1)                          (1:1,1:N1)

            Array Side Effects of Procedure SUB2 (inter-procedural)
VARIABLE         USE                               ASG
--------         ---                               ---
M               (1:N1,1:N2)
X               (1:N2)
Y               (1:N1)                             (K+1:K+N1)
```

Figure 17 Array table generated by `-show arrays`

Three columns show the names of the array variables in each procedure, which elements of each array are referenced in the procedure, and which elements of each array are assigned. This example shows the section of an array data table for procedures `sub1`, `sub2`, and `main`. The table indicates, for example, that procedure `sub1` references `M`(1,1) through `M`(`N2`,`N1`), `X`(1,1) through `X`(1,`N2`), and `Y`(1,1) through `Y`(1,`N1`). It assigns to `Y`(1,1) through `Y`(1,`N1`). In the section for `main`, asterisks indicate that the Application Compiler cannot determine the set of indices that have been referenced or assigned. The pound sign (#) next to the `Z` usage range indicates that this range is approximate; some elements between 1 and 100 may have been skipped.

Figure 17 shows an array table for Fortran procedures. The Application Compiler uses language-specific conventions to show arrays. Figure 18 shows the portion of the array table generated when `sub2` is a C-language procedure.

```
         Array Side Effects of Procedure main (inter-procedural)
VARIABLE         USE                                 ASG

--------         ---                                 ---
main:m           [1:n2][1:n1]
main:x           [1:n2]
main:y           [1:n1]                              [k+1:k+n1]
```

Figure 18 C-language array table

Here, the Application Compiler uses square brackets for arrays, and the subscripts for multidimensional arrays are in row-major order, as defined by the C language. The Application Compiler decides which format to use for an array based on the procedure in which the array appears. If a Fortran array is passed to a C procedure `dfun`, for example, the array appears as a C array inside `dfun`.

`-show calls`

The `-show calls` option causes the Application Compiler to produce a call graph as part of its output; this is illustrated in the following example.

In this example program, `MAIN` calls `MGEN`, `SEC`, `SB1`, and `LSUB`; `SB1` calls `foo1` and `XM1`. The call graph shows all Fortran procedure names in uppercase. For C, which is case sensitive, it shows procedure names in the original case. You cannot tell the difference between C procedure names written in uppercase and Fortran procedure names that are coerced to uppercase, but you can tell that names shown with lowercase characters, such as `foo1` in this example, are C procedures. The numbers on the left show the nesting level of a procedure.

```
0 MAIN
1    MGEN
1    SEC
1    SB1
2        foo1
2        XM1
1    LSUB
```

Note that this is a static call graph. Some of the calls shown may not be made at runtime. The special symbol "<<<" in a call graph indicates recursion. In the following example, it indicates a recursive call from baz back to foo.

```
0 main
1     foo
2          baz
3               <<< foo
1     bar
2          foo ...
```

Note also the "..." symbol following the second call to foo. This indicates that the second invocation of foo has the same subtree of calls as the first. Drawing the call graph for this program produces the picture shown in Figure 19.

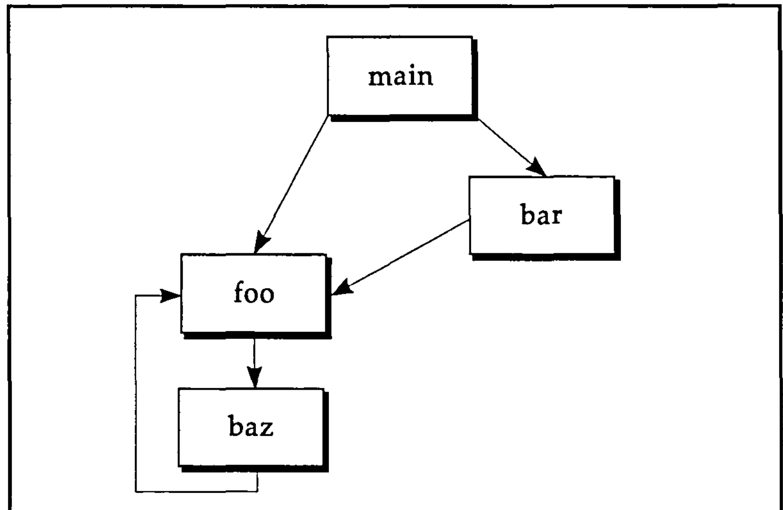


Figure 19 A call graph

The PDB Viewer can display graphical call graph illustrations like this; refer to Appendix B for more information.

-show clones

Invoking apc with the -show clones option produces an analyzer message every time a procedure is cloned. The following example shows a -show clones message for a Fortran procedure:

```
Created clone val$clone$1 of val for call
on line 82 of ./itgr.f to absorb constant
formal(s) IY=4, IX=3
```

This message shows that the procedure val is cloned once for the call on line 82 of itgr.f. Two constants are propagated into val$clone$1 to replace the formal parameters IY and

IX. Messages that indicate the cloning of procedures are interspersed with other analyzer messages. Thus, if val is cloned again from another procedure, you may see another message later on:

```
Created clone val$clone$2 of val for call
on line 28 of ./sax.f to absorb constant
formal(s) IY=3, IX=2
```

-show constants

The -show constants option provides detailed data about constants. Messages such as the following indicate that a constant has been propagated.

```
              Constants on Entry to Procedure SB1
SB1:LDA=301
              Constants on Entry to Procedure XM1
XM1:LDX=301
              Constants on Entry to Procedure MX1
MX1:LDM=301, MX1:K=0
              Constants on Entry to Procedure MGEN
MGEN:LDA=301
.
.
.
Compilation: Optimization: Propagated IPO constant on entry to `MGEN',
LDA = 301
```

The first portion of this report (before the ellipses) lists the constants that exist on entry to each procedure in the application. For example, the first line reports that the variable LDA, which is passed to SB1, has a value of 301 at the call to SB1. This portion reports every variable that is constant on entry to a procedure; not all of these variables will be propagated.

The "Propagated IPO constant" message appears immediately before the optimization report for MGEN and tells you that the optimizer actually did propagate the constant value 301 to MGEN. Such messages will immediately precede the optimization reports for every procedure into which constants were actually propagated. This information can help you spot variables that you expect to be constant but are not.

You may sometimes see messages reporting constants that do not appear in your program. These constants often have unusual names.

For example, compiling the following program

```
PROGRAM MAIN
CHARACTER*5 ST /"ABCDE"/
CALL SUB(ST)
END

SUBROUTINE SUB(A)
CHARACTER*5 A
PRINT *,A
END
```

produces this message:

```
          Constants on entry to procedure SUB
SUB: #A=5
```

The compiler generates a new dummy argument, #A, to represent the length of the dummy argument A. The interprocedural optimizer recognizes that the value passed to #A is a constant that can be propagated. Compiler-generated arguments appear most often in Fortran calls that pass CHARACTER arguments and in calls between Fortran and C.

-show inline

By default, the Application Compiler reports inlining as part of the optimization report. If you use -show inline, the Application Compiler also generates a separate set of messages that report all procedures automatically marked for inlining. Using these messages, you can determine which procedures were inlined automatically and which were inlined because of directives. Consider the following example, which shows messages generated by a program built using -show inline.

```
Optimization: Marked call from `MAIN' to
`FOO' on line 7 of ./inlinex.f for inlining

Optimization: Marked call from `MAIN' to
`FOO' on line 10 of ./inlinex.f for inlining

Optimization: Marked call from `MAIN' to
`FOO' on line 13 of ./inlinex.f for inlining
```

Procedure FOO is marked for inlining on line 7 of MAIN and, as the optimization report of Figure 20 shows, is automatically inlined (no directive caused this). The call to FOO at line 10 is marked and, according to the optimization report, not inlined due to the presence of a NO_INLINE_CALL directive. The call to FOO at line 13 is marked and, according to the optimization report, inlined.

```
             Optimization for Procedure MAIN

Line    Id          Reordering          New     Optimizing / Special
Num.    Num. Name   Transformation      Loops   Transformation
------------------------------------------------------------------------
   3      1  I      FULL VECTOR                 No Strip
   7         FOO    INLINE CALL Auto    See report for FOO (MAIN.FOO #1)
  10         FOO    No Inline
  13         FOO    INLINE CALL

           Optimization for Inline Copy of Procedure FOO (MAIN.FOO #1)

Line    Id          Reordering          New     Optimizing / Special
Num.    Num. Name   Transformation      Loops   Transformation
--------------------------------------------------------------------------
  19      1  J$1    FULL VECTOR                 No Strip
```

Figure 20 Optimization report showing inlined procedures

`-show outlines`

This option displays an outline of the loop and call structure for each procedure. A procedure with nested loops, for example, is outlined as shown in Figure 21.

```
               Outline of Procedure MX1

Begin
    loop J = 1 to N2 step 2
        loop I = ? to N1 step ?
        end loop
    end loop
End
```

Figure 21 Outline of procedure with loops

Whenever possible, the Application Compiler displays the constant value or variable name for the loop start, stop, and step values. You can see this in the outer loop. When it's not possible to determine a value or the value is too complicated to print, the Application Compiler substitutes a question mark (`?`). The inner loop shows these question marks for the start and step values.

Figure 22 shows the outline of a procedure with calls as well as loops. For each procedure call, the outline shows the arguments passed, the estimated frequency of the call

[freq=*n*], and an indication of whether the procedure being called is inlined [inline] or cloned [clone].

```
                 Outline of Procedure MAIN

Begin
   loop I = ? to 100 step ?
       call [inline] FOO(&W, 10) [freq=250]
   end loop
   call BAR(1) [freq=1.0]
   call [clone] BAR$clone$1(2) [freq=1.0]
   call [inline] GOO(&BAR) [freq=1.0]
End
```

Figure 22 Outline of procedure with loops and calls

-show pointers

The -show pointers option produces a table to show the targets found for pointers in each procedure when pointer tracking is enabled via the -enable pointer_track option. The table shows the pointer variables that are set and the targets they are set to, at the end (exit) of each procedure. In the example in Figure 23, by the time main exits, five pointers are set. However, only two distinct pointer targets, main:a and main:n, exist. Two sets of pointers reference these two memory locations; the pointers in each set are aliased together.

```
Interprocedural pointer ranges for
procedure main
NAME                    TARGETS
----                    -------
fodd:a                  main:a
fodd:b                  main:n
foo:a                   main:a
foo:b                   main:a
foo:n                   main:n

Interprocedural pointer ranges for
procedure foo
NAME                    TARGETS
----                    -------
foo:a                   main:a
foo:b                   main:a
foo:n                   main:n

Interprocedural pointer ranges for
procedure fodd
NAME                    TARGETS
----                    -------
fodd:n                  main:n
```

Figure 23 Pointer tables generated by `-show pointers`

The Application Compiler differentiates between automatic variables with the same name by prefixing each with the name of the procedure where it is declared. For example, `foo:n` is the pointer `n` declared in procedure `foo` (see the example code that follows), while `fodd:n` is the pointer `n` declared in procedure `fodd`.

```
fodd(a, n)
```

```
float *a;
int *n;
{
 int i, m;
 m= *n;

 for (i=0; i<m; i++)
   a[i]= n[i]+1;
}

foo(a,b,n)
float *a, *b;

{
 int i, m;

 for (i=100; i<900; i++)
   a[i]= (b[i-50]+b[i+50])/n[i];
}

main()
{
 float a[1000], b[1000];
 int n[1000];

 fodd(a,n);
 foo(a,b,n);
}
```

You may also see pointer variables in the array summary tables. Remember, in C, `p[i]` is equivalent to `*(p+i)`.

`-show renames`

To prevent conflicts caused by interprocedural optimizations, the Application Compiler frequently must rename variables. Using a debugger such as CXdb, however, requires that you know the real names of variables. The `-show renames` options provides information about any variables that the Application Compiler has renamed.

Whenever the Application Compiler makes a copy of a procedure, it must rename static variables in each copy to prevent name conflicts. The Application Compiler makes copies of procedures for cloning, for inlining, and for handling Fortran multiple entries.

The messages contained in Figure 24, for example, show that the variable `I` in `MGN` was renamed because of inlining. Variables `A` and `B` in `FOO`, `BAZ`, and `BAR` are renamed because

Using the APC

those procedures have multiple entries, and variable H in FOO and FOO$clone$1 is renamed because of cloning.

```
Advisory: Renamed local static variable I in routine MGN to
I$MGN$inline$5
Advisory: Renamed local static variable A in routine FOO to
A$FOO$entry$0
Advisory: Renamed local static variable B in routine BAZ to
B$FBAZ$entry$0
Advisory: Renamed local static variable B in routine BAR to
B$BAR$entry$0
Advisory: Renamed local static variable H in routine FOO to
H$FOO$clone$1
Advisory: Renamed local static variable H in routine FOO$clone$1
to H$FOO$clone$1
```

Figure 24 Renamed variable messages

The Application Compiler also creates new names to handle file-scoped static variables and functions in C. The Application Compiler must do this to fit these "semiglobal" variables into an internal representation that understands only local and global scope. Semiglobals are changed to globals and renamed to prevent name conflicts. In Figure 25, the static variable sV is renamed.

```
Advisory: Renamed file-scope static variable sV in file t.c to
sV$t$0
```

Figure 25 Renamed static variable message

-show resizes

The -show resizes option tells the Application Compiler to issue a message when an array is resized, such as:

```
Optimization: Marked global array
'/BLOCK/A' for resizing
```

or

```
Optimization: Marked array 'A' in procedure
'FOO' for resizing
```

-show scalars

The -show scalars option tells the Application Compiler to display a list of scalar variables referenced and assigned in each procedure. The example in Figure 26 shows the use, assign and kill lists for the procedure FOO. The argument ARGZ and COMMON-block variable X in BLOCK are used, meaning the procedure may read them, and assigned,

meaning the procedure may assign them; the COMMON-block variable Y in BLOCK2 is used but not assigned; X in BLOCK is killed, meaning the procedure definitely assigns it on every call. Argument names are prefaced with the procedure name to eliminate ambiguity when variables of the same name appear in more than one procedure.

```
Scalar Side Effects of Procedure FOO

USES: /BLOCK/X, /BLOCK2/Y, FOO:ARGZ
ASGS: /BLOCK/X, FOO:ARGZ
KILLS: /BLOCK/X
```

Figure 26 Use and assign lists generated using `-show scalars`

`-show state`

Displays the current state of the program database. For example, assume the following command line has been executed:

```
apc -upto analysis
```

Executing the following command line:

```
apc -show state
```

would then yield:

```
Completed analysis phase
```

This option is useful if you are unsure of the last phase built. Refer to the "Partial build options" section for more information.

`-show symbols`

The `-show symbols` option tells the Application Compiler to display the program symbol table. Figure 27 shows the symbol table output for one Fortran subroutine. (See the section on `-show types` in this chapter for details on interpreting type information.)

```
                    Global Symbol Table

Subroutine:         SUBA     (declared in ./main.f)
        SUBA:A      type:       REAL*4(1000)
                    aliases:

        SUBA:B      type:       REAL*4(1000)
                    aliases:

        SUBA:N      type:       INTEGER*4
                    aliases:
```

Figure 27 Subroutine symbol table

Figure 28 shows the symbol table output for a Fortran function.

```
Function:           SUM      (declared in ./main.f)
        SUM:X       type:    REAL*4(1000)
                    aliases:
```

Figure 28 Function symbol table

Figure 29 shows the symbol table output for a Fortran COMMON block.

```
Common Block:  /CM/   (declared in ./common.f)     size:   8004 bytes
        /CM/A    type:      REAL*4(1000)
                 offset:    0
                 aliases:
                 value:     0
        /CM/IA   type:      INTEGER*4(1000)
                 offset:    4000
                 aliases:
                 value:     0
        /CM/IINT        type:      INTEGER*4
                        offset:    8000
                        aliases:
```

Figure 29 COMMON block symbol table

`-show types`

The `-show types` option tells the Application Compiler to display additional information on procedure calls with type problems. Figure 30 shows the messages generated by `-check types` and `-show types` for a single type mismatch. `-check types` generates the warning message shown first; `-show types` gives information on the types

passed in the call on line 16 of t.f and the types expected according to the definition of SB1 on line 1 of sb1.f.

```
Warning: Argument number 1 of DFUN has inconsistent type in ./t.f on
line 4 and ./t.f on line 8
Warning: Argument number 2 of DFUN has inconsistent type in ./t.f on
line 4 and ./t.f on line 8
Call to DFUN on line 4 of ./t.f: REAL*4 ((INTEGER*4 *), (INTEGER*4 *))
Defn of DFUN on line 8 of ./t.f: REAL*4 ((REAL*4 *), (REAL*4 *))
Warning: Argument number 2 of SB1 has inconsistent type in ./t.f on
line 16 and ./sb1.f on line 1
Call to SB1 on line 16 of ./t.f: VOID ((REAL*8 *), (REAL*4 *))
Defn of SB1 on line 1 of ./sb1.f: VOID ((REAL*8 *), (INTEGER*4 *))
```

Figure 30 Messages generated with -check types and -show types

The -show types option shows all procedures as functions. The function's return type appears before the outer parentheses, which enclose a list of types corresponding to the parameter or argument list.

The function DFUN, for example, returns a REAL*4 value. Fortran subroutines and C functions that do not return a value have a return type called VOID or void. Fortran types appear in uppercase; C types appear in mixed or lowercase. The call on line 16 of t.f passes two arguments to SB1, one of type (REAL*8 *) and one of type (REAL*4 *). Commas separate the two types. The extra asterisk following REAL*8 indicates a pointer type. This notation may seem confusing, since Convex Fortran does not support pointers, but there is a reason for it.

Programming languages can pass arguments by value or by reference. Pass-by-value makes a copy of the argument and passes that copy to the procedure being called. The procedure can modify only the local copy (that is, the dummy argument or formal parameter); the original copy remains safe. Pass-by-reference sends the address of the argument, so any changes made by the called procedure affect the original argument. C passes scalar arguments by value. Fortran, on the other hand, passes arguments by reference. So, when Fortran passes an argument to a subroutine, it is really passing an invisible pointer or address value. To simplify interlanguage programming, the Application Compiler makes these invisible pointers visible.

The Application Compiler adds a set of parentheses to reinforce the association between a type and the asterisk. So, (REAL*4 *) in the argument list means that the variable

passed in that position has type REAL*4 and is passed by reference.

When a Fortran procedure calls a C function, the Application Compiler shows the Fortran types for the call and C types for the definition.

Consider the following program fragments.

Fortran example:

```
PROGRAM MAIN
DOUBLE PRECISION Y(1000), M(300,1000)

DO I=1,301
CALL X1(I, Y, M)
ENDDO
```

C Example:

```
X1_(n1, y, m)
int *n1;
double y[], m[1000][300];
```

For this code, the -show types option produces the messages show in Figure 31.

```
Call to X1 on line 5 of ./main.f:VOID((INTEGER*4 *),(REAL*8 *),
(REAL*8 *))
Defn of X1 on line 7 of ./x1.c: int ((int *), (double *),
(double[0:299] *))
```

Figure 31 -show types messages generated when Fortran main calls C function

The first argument passed to X1 is a Fortran INTEGER*4. C declares this as (int *). The second argument in the definition of X1 has type (double *). This is either a pointer to a double or an array with one dimension. C passes arrays as pointers, whether the programmer explicitly requests it or not, so pointers and one-dimensional arrays have the same appearance.

The third argument is a two-dimensional array. In the definition message, this appears as a pointer to a one-dimensional array. (Because C passes pointers as arrays, a pointer to an array is equivalent to a two-dimensional array.) [0:299] following the type (double) shows the second dimension of the array in C (equivalent to the first dimension in Fortran). On the Fortran side, this appears simply as (REAL*8).

Differences in the way the languages are compiled prevent the Application Compiler from showing the dimensions of Fortran arrays for either calls or definitions.

The messages in Figure 32 show that t.f passes an INTEGER*1 array and an INTEGER*4 scalar to the DATE function. The INTEGER*1 data type in Fortran corresponds to the char type in C. Again, the asterisk indicates that the first argument is passed as a pointer to the C function. Note the type mismatch in the return value. The C code declares DATE as a function that returns int, but the Fortran code in t.f uses it as a subroutine.

```
Call to DATE on line 3 of ./t.f: VOID ((INTEGER*1 *), INTEGER*4)
Defn of DATE in /usr/convex/ipo/lib/libU77.a: int ((char *), (int *))
```

Figure 32 -show types messages generated when Fortran main calls DATE function

Some Fortran data types, such as COMPLEX, do not have equivalent built-in types in C. For further information on equivalences between Fortran and C data types, consult the *Convex Interlanguage Programming Guide*. Figure 33 shows how a Fortran call passes a COMPLEX*16 data type to a C function, which declares it as a user-defined structure.

```
Call to FUN2 on line 13 of ./main.f: REAL*4 ((COMPLEX*16 *))
Defn of FUN2 on line 6 of ./fun2.c: int (struct{double r; double i;})
```

Figure 33 -show types messages generated when passing COMPLEX data type to C function

Language structures sometimes make type information unavailable to the Application Compiler. When this happens, the Application Compiler prints NO INFO in the appropriate field.

-show all

The -show all option causes the Application Compiler to show all of the tables. Running apc -show all has the same effect as running apc with the following -show options:

- aliases
- arrays
- calls
- constants
- clones
- inline
- outlines
- pointers

- renames
- resizes
- scalars
- state
- symbols
- types

-u or -help

The options cause the Application Compiler to display a list of all possible apc options and their usage.

-v

The -v, or verbose, option causes the Application Compiler to print the names of each procedure during the analysis and compile phases, and the names of the interprocedural algorithms during the synthesis phase. Note that -s and -v are mutually exclusive. Figure 34 shows the messages that apc generates when compiling the file shown in the last two examples, this time with -v. (The optimization and IPO reports are abbreviated for clarity.)

```
% apc -v
Computing Dependencies
Summary phase
Summarizing './mx1.f' (-O2)
Summarizing './xm1.f' (-O2)
Summarizing './sb1.f' (-O2)
Summarizing './main000.f' (-O2)
Summarizing './mgen.f' (-O2)
Summarizing './sec.f' (-O2)
Synthesis phase (pass 1)
Performing library call resolution
Performing type checking
Performing call analysis
Performing alias analysis
Performing pointer tracking
Analysis phase
Analyzing procedure 'MX1'
Analyzing procedure 'XM1'
Analyzing procedure 'MAIN'
Analyzing procedure 'MGEN'
Analyzing procedure 'SEC'
Synthesis phase (pass 2)
Performing scalar analysis
Performing constant propagation
Performing inline analysis
Performing clone analysis
Performing array analysis
Performing error analysis
Compilation phase
Compiling procedure 'MX1'
Compiling procedure 'XM1'
<Opt Report for XM1>
Compiling procedure 'MAIN'
Compilation: Warning on line 6.27 of ./sec.f: 'T' used before being
assigned.
Compilation: Warning on line 10.3 of ./sec.f: 'T' used before being
assigned.
<Opt Report for MAIN>
<Opt Report for MAIN.MGEN (Inline copy of MGEN in MAIN)>
Compiling procedure 'MGEN'
<Opt Report for MGEN>
Compiling procedure 'SEC'
Link phase
Linking 'Lin1' (-L/usr/convex/ipo/lib -lveclib )
<IPO Report>
```

Figure 34 APC messages with -v option

Optimization options

The options listed in this section control optimizations performed by the Application Compiler.

`-assert` *subscripts_ok*

This option makes an assertion about the validity of your program's subscripts. This may allow the Application Compiler to extend the dimension of some arrays that otherwise could not be extended, reducing the potential for memory-bank conflicts that slow a program down.

`-clone`

The `-clone` option has two possible arguments: `all` and `none`. The following examples show how it is used.

```
% apc -clone all
% apc -clone none
```

The `-clone` option provides a functionality similar to the `CLONE` and `NO_CLONE` directives described in Chapter 6, but without the need to modify source code by adding directives. The `-clone all` option has the same effect as applying a `CLONE` directive to every procedure. The `-clone none` option has the same effect as applying a `NO_CLONE` directive to every procedure. `-clone none` prevents the Application Compiler from cloning any procedure. `-clone all` tells the Application Compiler to clone a procedure whenever cloning allows it to replace any argument by a propagated constant—even an argument that the Application Compiler does not consider important by default.

See the sections on the `CLONE` and `NO_CLONE` directives in Chapter 6 for more information.

`-enable pointer_track`

The `-enable pointer_track` option enables pointer tracking, which is described in detail in Chapter 2.

`-extend_dim`

The `-extend_dim` option tells the Application Compiler how to optimize array storage.

On C Series machines, array-storage optimization seeks to eliminate bank conflicts. Bank conflicts occur when a nonmajor dimension of an array (i.e. a dimension other than the rightmost dimension in Fortran or the leftmost dimension in C) is a multiple of the number of memory banks, or interleave, on your machine.

The C Series Application Compiler reduces bank conflicts by extending the problem dimensions of an array when this

condition occurs. Storage optimization extends the troublesome dimensions of an array to

1+(*interleave**(1+*ol*/*interleave*))

where *ol* is the original length of the dimension and the division is integer division.

The rightmost dimension of Fortran arrays and leftmost dimension of C arrays do not cause problems, so single-dimensional arrays are never extended.

On SPP Series machines, array storage optimization seeks to eliminate contention for cache lines. This is done by insuring that nonmajor array dimensions are integrally divisible by the number of array elements that fit into a cache line. For example, consider the following Fortran array:

```
REAL*4 X(65,64)
```

Recall that an SPP Series processor cache line is 32 bytes in size, so it can hold 8 `REAL*4` data items. 65 is not integrally divisible by 8, so the leftmost dimension of `X` is increased to 72.

When combined with SPP Series data localization optimizations, array storage optimizations decrease the likelihood that two processors will attempt to access the same cache line at once.

Array storage optimizations are performed by default at optimization level `-O3` on SPP Series machines. The `-extend_dim multiple` or `-extend_dim all` options are not available on the SPP platform, but the optimization can be disabled using the `-extend_dim none` option.

On C Series machines, you can select the level of storage optimization you want the Application Compiler to perform:

- If you specify `-extend_dim multiple`, the Application Compiler extends only those dimensions that are exact multiples of the interleave. This is the default.
- If you specify `-extend_dim all`, the Application Compiler extends dimensions that are not equal to *1+interleave*N*, where *N* is an integer greater than or equal to zero.
- If you specify `-extend_dim none`, dimensions are not extended. This option also works on SPP Series machines, to disable automatic array storage optimizations.

C Series only

Table 3 shows the effects of the `-extend_dim` option on Fortran arrays for an *interleave* value of 32:

Table 3 -extend_dim effects on Fortran arrays (C Series only)

Original array	-extend_dim none	-extend_dim multiple	-extend_dim all
A(60)	A(60)	A(60)	A(60)
A(64)	A(64)	A(64)	A(64)
A(64,64)	A(64,64)	A(97,64)	A(97,64)
A(60,64)	A(60,64)	A(60,64)	A(65,64)
A(60,64,64)	A(60,64,64)	A(60,97,64)	A(65,97,64)

C Series only

Table 4 shows the effects of the -extend_dim option on arrays in C:

Table 4 -extend_dim effects on C arrays (C Series only)

Original array	-extend_dim none	-extend_dim multiple	-extend_dim all
A[60]	A[60]	A[60]	A[60]
A[64]	A[64]	A[64]	A[64]
A[64][64]	A[64][64]	A[64][97]	A[64][97]
A[64][60]	A[64][60]	A[64][60]	A[64][65]
A[64][64][60]	A[64][64][60]	A[64][97][60]	A[64][97][65]

The Application Compiler cannot extend the dimension of the following arrays on either platform:

- Arrays that are in a Fortran equivalence with other arrays of different dimension or element size
- Arrays that are passed as arguments with reshaping (a change in dimension from one procedure to another)
- Arrays that are input or output as a whole (Arrays that are input or output by elements are okay.)
- Arrays that are accessed with unsafe subscripts, unless the -assert subscripts_ok option is used.

-inline *level*

The -inline option tells the Application Compiler how aggressively to try to inline your code. By default, the Application Compiler chooses a moderate level of inlining.

Using the -inline option, you can change the amount of inlining that the Application Compiler attempts, as shown in Table 5.

Table 5 Levels of inlining

Option	Function
-inline low	Inline fewer procedures
-inline medium	Default inlining
-inline high	Inline more procedures
-inline none	Inline no procedures

-link_sort

The -link_sort option tells the Application Compiler which link optimization algorithm to use. This option has three possible arguments, as shown in Table 6.

Table 6 -link_sort options

Option	Function
-link_sort none	No link optimization
-link_sort times_called	Sorts objects based on the APC's estimate of the number of times each procedure is called (default).
-link_sort percent_time	Sorts objects based on the APC's estimate of the percentage of execution time spent in the procedure.

This optimization reduces page faults and improves instruction-cache performance. You can improve the estimates, and thus the effectiveness of this algorithm, by providing profile data with the -prof or -gprof option on C Series machines or with the -pdf option on both platforms.

To instruct the Application Compiler to use a different link-sorting algorithm, based on the time spent in each procedure, use -link_sort percent_time. This algorithm requires profile data, so you must also use the

-prof or -gprof option and have a previously prepared mon.out or gmon.out file. (The gprof profiler provides better information than prof, so using gprof will produce better link optimization.) To disable link optimization, use the -link_sort none option.

Partial build options

The options listed in this section control the phases of compilation performed by the Application Compiler, allowing you to partially build applications on a phase-by-phase basis. You can use the pdbview utility to examine program databases from partially built applications between compilation phases; this provides the opportunity to check the results of each phase.

-upto *phase*
Allows partial building up to and including the phase specified in *phase*. Table 7 shows valid values for phase.

Table 7 *phase* arguments

phase	**Builds to**
summary	Summary phase
synth1	First synthesis phase
analysis	Analysis phase
synth2	Second synthesis phase
compilation	Compilation phase
link	Link phase

-upto can be combined with -cont to continue a partial build from a previously-completed partial build.

-cont
Continues a previously-completed partial build. You can stop the build at any remaining phase by adding an -upto option, as shown:

```
apc -cont -upto synth2
```

This command line assumes a partial build has already been performed up to a phase that precedes the second synthesis phase. On execution of this command line, building would continue up to the second synthesis phase. If you do not specify an -upto with the -cont, building will proceed to completion.

The -show state option, which is described in the "Message control options" section, allows you to determine the last compilation phase completed.

Program database options

The options listed in this section control various aspects of the Program Data Base.

`-c`

The `-c`, or cleanup, option removes the program database (PDB). When you build a program, the Application Compiler retains the program database so that the entire program need not be analyzed and compiled again when a single procedure is changed.
To clean up the database, use the `-c` option with the `apc` command:

```
% apc -c
```

When the `-c` option is used, the `apc` command does not create an executable. Except for `-path`, any other options used with the `-c` option are ignored.

If the program database becomes corrupted so that `apc -c` cannot remove it, remove the database manually by using the `rm` command:

```
% rm -r PDB
```

If the execution of `apc` is interrupted by a system failure, user interrupt, or any other cause, use the `-c` option to remove the partially constructed database, which may contain inconsistent information.

`-compress`

The `-compress` option causes the Application Compiler to compress `PDB` data. This reduces disk space usage by an average of 30%, but can also lengthen compile times.

`-path`

The `-path` option specifies a path (directory name) where the Application Compiler is to put the program database. If you do not use the `-path` option to specify a path name, the database (which is always named `PDB`) is placed in the current directory by default.

Profiling options

The Application Compiler can use the output of a profile to acquire better information about the frequency and duration of

procedure calls. This allows the Application Compiler to make better decisions about inlining and link optimization. This section lists the profiling options available for use with apc.

Note

The gprof and prof profilers are not available on SPP Series machines.

C Series only

-gprof

-gprof allows Application Compiler to use profiling information generated by gprof, the Convex C Series graph profiler. To use this option, you must:

1. Compile your program into an executable using the appropriate procedural compiler with the -pg option.
2. Run the resulting executable to produce a gmon.out monitor file.
3. Invoke apc with the -gprof option and the name of the executable produced.

An example follows.

```
% fc -pg *.f -o /mnt/me/gprof.out
% gprof.out
% apc -gprof /mnt/me/gprof.out
```

When invoked, the APC builds the application (using the buildfile in the current directory) as usual, but uses the profiling data contained in gmon.out to improve optimization.

C Series only

-prof

-prof allows the Application Compiler to use profiling information generated by prof, the standard Convex C Series profiler. To use this option, you must:

1. Compile your program into an executable using the appropriate procedural compiler with the -p option.
2. Run the resulting executable to produce a mon.out monitor file.
3. Invoke apc with the -prof option and the name of the executable produced.

An example follows.

```
% fc -p *.f -o /mnt/me/prof.out
% prof.out
% apc -prof /mnt/me/prof.out
```

(The gprof profiler provides better information than prof, so using gprof will produce better link optimization.)

When invoked, the APC builds the application (using the buildfile in the current directory) as usual, but uses the

profiling data contained in mon.out to improve optimization.

-pdf

-pdf allows the Application Compiler to use profiling information generated by CXpa, the Convex Performance Analyzer, which is available on both SPP and C Series machines. To use this option, you must:

1. Compile your program into an executable using the appropriate procedural compiler with the -cxpa or -cxpar option (-cxpab does not provide enough information to be useful to the APC).
2. Run CXpa with the resulting executable to produce a .pdf monitor file.
3. Invoke apc with the -pdf option and the name of the .pdf file produced.

An example follows.

```
% fc -cxpa *.f -o /mnt/me/perf.out
% cxpa -nw /mnt/me/perf.out

(CXpa) sel all
(CXpa) run
(CXpa) quit

% apc -pdf /mnt/me/perf.out.pdf
```

Commands preceded by "%" are typed at the shell command line; those preceded by "(CXpa)" are typed within CXpa. CXpa is invoked in text mode (-nw) in this example to simplify the illustration, but could also be used in X-window mode.

When invoked, the APC builds the application (using the buildfile in the current directory) as usual, but uses the profiling data contained in perf.pdf to improve optimization.

Creating the initial executable with the APC (rather than fc or cc) is not recommended, because the optimizations performed by the APC make it difficult to map the profile information back to the original source.

Optimization report

When the Application Compiler inlines or clones a procedure, it generates additional optimization reports for the cloned or inlined copies of the procedure. Consider the following Fortran procedure, which is inlined twice into `MAIN`:

```
      PROGRAM MAIN
      REAL A(100)
      DO I=1, 100
        A(I) = I
      ENDDO

C$DIR INLINE_CALL
      CALL FOO(1)

C$DIR INLINE_CALL
      CALL FOO(-1)
      END

      SUBROUTINE FOO(ISTEP)
      REAL A(100)
      DO J=1,50,ISTEP
        A(J)=A(J)+1
      ENDDO
      END
```

Instead of the single optimization report for procedure `FOO` generated by the conventional Fortran compiler, the Application Compiler generates three separate reports. As shown in Figure 35, the first report shows the optimizations performed on the procedure `FOO` itself. Two additional reports show the optimizations performed on `MAIN` and on the inline copy of `FOO` that the Application Compiler inlines into `MAIN`. The optimization report for `MAIN` is modified to show the source-code line numbers where these two copies were inlined.

Note

All example optimization reports presented in this chapter were generated on C Series machines; SPP Series optimization reports are similar except for references to vectorization.

```
          Optimization for Procedure FOO

Line      Id                Reordering       New     Optimizing / Special
Num.      Num.  Name        Transformation   Loops   Transformation
------------------------------------------------------------------------
  16        1   J           FULL VECTOR

           Optimization for Procedure MAIN

Line      Id                Reordering       New     Optimizing / Special
Num.      Num.  Name        Transformation   Loops   Transformation
------------------------------------------------------------------------
   3        1   I           FULL VECTOR No Strip
   8            FOO         INLINE CALL   See report for FOO (MAIN.FOO #1)
  11            FOO         INLINE CALL

           Optimization for Inline Copy of Procedure FOO (MAIN.FOO #1)

Line      Id                Reordering       New     Optimizing / Special
Num.      Num.  Name        Transformation   Loops   Transformation
------------------------------------------------------------------------
  16        1   J$1         FULL VECTOR              No Strip
```

Figure 35 APC optimization reports

The optimization report also explains whether a call is inlined automatically or because of a directive. In Figure 35, both calls to FOO were inlined as the result of directives. The partial optimization report in Figure 36 shows two calls to a procedure that are inlined automatically. Note that these calls to procedure SFUN are marked `INLINE CALL Auto` in the Reordering Transformation column.

```
           Optimization for Procedure MAIN

Line     Id            Reordering        New     Optimizing / Special
Num.     Num.   Name   Transformation    Loops   Transformation
----------------------------------------------------------------------
   3       1    I      FULL VECTOR               No Strip
   8            FOO    INLINE CALL Auto See report for FOO (MAIN.FOO #1)
  11            FOO    INLINE CALL Auto
```

Figure 36 Optimization report for automatically inlined procedure

If a procedure call is not inlined even though it is marked for inlining by a directive, the optimization report tells you the reason why the call isn't inlined. If the Application Compiler determines that a call is profitable to inline but cannot inline the call, once again, the optimization report tells you why. Figure 37 shows the optimization report for a program where procedure FOO might be inlined into MAIN at lines 7 and 10. Errors in the number and type of arguments passed to FOO prevent this inlining.

```
            Optimization for Procedure FOO

Line       Id                Reordering       New     Optimizing / Special
Num.       Num.  Name        Transformation   Loops   Transformation
------------------------------------------------------------------------
  19          1  J           FULL VECTOR

            Optimization for Procedure MAIN

Line   Id               Reordering       New     Optimizing / Special
Num.   Num.  Name       Transformation   Loops   Transformation
------------------------------------------------------------------------
   3      1  I          FULL VECTOR              No Strip
   7         FOO        No Inline
  10         FOO        No Inline

Line   Id    Iter.      Analysis
Num.   Num.  Var.
------------------------------------------------------------------------
   7         FOO       Mismatched number of arguments
  10         FOO       Type mismatch of argument #1
```

Figure 37 Optimization report for procedure that can't be inlined

The Application Compiler also modifies the optimization report for each procedure clone. Consider, for example, the following code:

```
      PROGRAM MAIN

      REAL A(200)
      CALL FOO(A, 100)
      CALL FOO(A, 200)
      END
```

```
      SUBROUTINE FOO(A, N)
C$DIR NO_INLINE
      INTEGER N
      REAL A(N)
      DO I=1,N
        A(I)=I
      ENDDO
      END
```

The optimization reports generated by this code are shown in Figure 38. To optimize the DO loop at line 11 in procedure FOO, the Application Compiler creates a clone of the procedure. This allows the Application Compiler to propagate the two integer constants, 100 and 200, into FOO and clone 1 of FOO (FOO$clone$1). The optimization report for MAIN shows that the clone is called on line 4.

```
           Optimization for FOO

Line      Id            Reordering         New     Optimizing / Special
Num.      Num.  Name    Transformation     Loops   Transformation
------------------------------------------------------------------
  11         1  I       FULL VECTOR                No Strip

            Optimization for FOO$clone$1

Line      Id            Reordering         New     Optimizing / Special
Num.      Num.  Name    Transformation     Loops   Transformation
------------------------------------------------------------------
  11         1  I       FULL VECTOR

            Optimization for MAIN

Line      Id            Reordering         New     Optimizing / Special
Num.      Num.  Name    Transformation     Loops   Transformation
------------------------------------------------------------------
   3            FOO     No Inline
   4            FOO     CLONE CALL       See report for clone #1 of FOO

Line     Id     Iter.  Analysis
Num.     Num.   Var
----------------------------------------------------------------------
   3            FOO    Automatic inlining prevented by no_inline directive
   4            FOO    Target of call changed from FOO to FOO$clone$1
```

Figure 38 Optimization report for cloned procedure

IPO report

Figure 39 shows the format of the interprocedural optimization (IPO) report.

Optimizations performed

Procedure	Times Procedure Inlined	Times Procedure Cloned	Propagated Constants Used	Pointer Variables Renamed	Arrays Resized
XM1			1		
SEC	2				
MX1			2		
LUS			1		
MAIN					
MGEN	1		1		
SB1			1		
Totals	3		6		
apc -show	inline	clones	constants	pointers	resizes

Errors Detected

Procedure	Mis-Matched Arg	Wrong Number Of Args	Mis-Matched Return Type	Invalid Aliases	Scalar Passed To Array	Invalid SubScript	Variables Not Initialized
XM1				12			
SEC							
MX1				6			
LUS							
MAIN							
MGEN							
SB1							
Totals				18			
apc -check	types	types	types			arrays	init

Figure 39 IPO report

The IPO report has two tables, which list optimizations performed and errors detected by interprocedural analysis. The first table has columns for procedures inlined and cloned, constants propagated, pointer classes formed, and arrays resized. Each row shows the number of optimizations performed in each of the categories for one procedure. Columns are labeled at both top and

bottom. The top label shows the type of optimization performed; the bottom label shows the `-show` option you use to request further information on that particular optimization.

The numbers in the "Times Procedure Inlined" and "Times Procedure Cloned" columns show the number of times that the procedure named on the left was cloned or inlined, not the number of calls cloned or inlined within the procedure. The "Propagated Constants Used" column shows the number of times that constants propagated into a procedure are used. Since each propagated constant can, potentially, be used multiple times, this number may be much greater than the number of constant propagation messages.

The second column from the right, "Pointer Variables Renamed," deserves special mention. This column shows the number of pointer variables that the Application Compiler renames to create unique pointer classes. Each pointer class has its own set of targets, which are not referenced by another pointer class. Aliasing, therefore, can occur within a pointer class, but not between classes. In general, the more pointers that are renamed, the more pointer classes that are found, and the less a program suffers from aliasing. You can use this column, then, as an indication of the Application Compiler's effectiveness in preventing alias problems in your program.

The rightmost column, "Arrays Resized," shows the number of arrays resized to prevent bank or cache line conflicts. Numbers in this column show the number of arrays resized in each procedure.

The second table has columns for mismatched argument types in procedure calls, wrong number of arguments used in procedure calls, mismatched return type for function calls, invalid aliases within a procedure, scalars passed to a procedure call that expects an array, invalid subscripts within a procedure, and variables used within a procedure without being initialized.

The "Invalid Aliases," "Invalid Subscripts," and "Variables Not Initialized" columns refer to problems found within the procedure named on the left. The columns that refer to arguments and return types report information on calls from the procedure named on the left. The labels at the bottom of the table show the `-check` options you use to request further information on a particular error type.

Creating a buildfile

4

The apc command is the Application Compiler's visible interface. Information required to compile an application is specified in a file called a buildfile. Although analogous in concept to a makefile, the buildfile is quite different in structure and function. Instead of using the buildfile to specify compilation dependencies that determine the order in which files must be compiled, apc scans the source files to get this information. This makes buildfiles simpler and easier to write; in many cases, they need include nothing more than a list of compiler and linker options.

A simple buildfile

A buildfile is an ASCII text file of two or more lines. You can create a buildfile using any ASCII text editor. Figure 40 shows a simple buildfile with two actual statements and four comment lines. (A comment line begins with the # symbol.)

```
# this line specifies the file to be
# compiled and the compiler options to be used:
source.f -cxdb -pdf -O2

# the next line specifies the standard library
# that the compiler is to link the program to:
link Fortran -cxdb -pdf
```

Figure 40 Example buildfile

The apc command assumes that a source file whose name ends in .f is a Fortran file and a source file whose name ends in .c is a C file. Each source file name must be on a separate line.

A list of compiler options can follow each file name specified. The buildfile shown in Figure 40 compiles the file source.f with debugging and profiling options at optimization level -O2.

Buildfiles

The Application Compiler recognizes most options accepted by the conventional Convex C and Fortran compilers. The Application Compiler does not support the conventional compiler's `-c` option, which suppresses the linking of object files. Do not use the `-c` option in a buildfile. Do not specify object files or libraries on the same line with source files.

The Application Compiler does not support several compiler options in buildfiles. For a complete list of unsupported compiler options, see Appendix A. For complete descriptions of the compiler options, see the *Fortran User's Guide* or *C User's Guide*.

The `link` line specifies object files and libraries to be linked into the compiled program, along with link options, if any. Specifying `Fortran` here links in all Fortran libraries in the standard order. If you are compiling a C program, use `C` instead. One of these two options must be specified, or no finished executable can be produced.

The Application Compiler must have the source for your program's main procedure. You cannot compile the main procedure with another compiler and link the resulting object file into a program compiled with the Application Compiler.

The Application Compiler does not accept assembly-language `.s` files on a compile line. To include assembly language in your programs, assemble the files first, then include the resulting `.o` files on the `link` line. For more information, refer to the "Using psum directives" section in Chapter 5.

Buildfile statements

A buildfile can include certain optimization statements that provide an alternative to the directives discussed in Chapter 6. Use these statements instead of directives when you want to avoid making changes to your source files.

You can add any of the following optimization statements to your buildfile:

- `inline`(*proclist*)
- `no_inline`(*proclist*)
- `clone`(*proclist*)
- `no_clone`(*proclist*)
- `no_alias`(*varlist*)

Each of these statements must appear on a separate line. Placing any of these statements in the buildfile has the same effect as placing the corresponding directive after the procedure definition in the source file (except `no_alias`, which has no corresponding

directive). See Chapter 6, "Directives", for a complete description of each directive's meaning.

Table 8 shows some example uses of the `inline` and `no_inline` statements and their directive equivalents. The examples in the column on the left show the optimization statement as it appears in the buildfile. The center column shows how a directive or directives would be placed in a Fortran source file to achieve the same effect. The column on the right shows how the directives would be used in a C source file.

Table 8 `inline/no_inline` statements and equivalents

Buildfile statement	Fortran directive	C pragma
`inline(FOO)`	`FUNCTION FOO` `C$DIR INLINE`	`FOO()` `#pragma _cnx inline`

`inline(FOO1,FOO2,FOO3)`	`FUNCTION FOO1`
	`C$DIR INLINE`
	`...`
	`FUNCTION FOO2`
	`C$DIR INLINE`
	`...`
	`FUNCTION FOO3`
	`C$DIR INLINE`
		`FOO1()`
		`#pragma _cnx inline`
		`...`
		`FOO2()`
		`#pragma _cnx inline`
		`...`
		`FO3()`
		`#pragma _cnx inline`

`no_inline(BAR)`	`FUNCTION BAR` `C$DIR NO_INLINE`	`BAR()` `#pragma _cnx no_inline`

`no_inline(BAR1,BAR2,BAR3)`	`FUNCTION BAR1`
	`C$DIR NO_INLINE`
	`...`
	`FUNCTION BAR2`
	`C$DIR NO_INLINE`
	`...`
	`FUNCTION BAR3`
	`C$DIR NO_INLINE`
	`...`
		`BAR1()`
		`#pragma _cnx no_inline`
		`...`
		`BAR2()`
		`#pragma _cnx no_inline`
		`...`
		`BAR3()`
		`#pragma _cnx no_inline`
		`...`

Table 9 shows some uses of the `clone` and `no_clone` statements and their directive equivalents.

Table 9 `clone/no_clone` statements and equivalents

Buildfile statement	Fortran directives	C pragma
clone(FOO)	FUNCTION FOO C$DIR CLONE	FOO() #pragma _cnx clone
clone(FOO1,FOO2,FOO3)	FUNCTION FOO1 C$DIR CLONE ... FUNCTION FOO2 C$DIR CLONE ... FUNCTION FOO3 C$DIR CLONE	FOO1() #pragma _cnx clone ... FOO2() #pragma _cnx clone ... FOO3 #pragma _cnx clone
no_clone(BAR)	FUNCTION BAR C$DIR NO_CLONE	BAR() #pragma _cnx no_clone
no_clone(BR1,BR2,BR3)	FUNCTION BR1 C$DIR NO_CLONE ... FUNCTION BR2 C$DIR NO_CLONE ... FUNCTION BR3 C$DIR NO_CLONE	BR1() #pragma _cnx no_clone ... BR2() #pragma _cnx no_clone ... BR3 #pragma _cnx no_clone

The `no_alias` statement tells the Application Compiler to ignore a potential alias between two variables. The two variables can consist of a Fortran dummy argument and a COMMON-block variable, two Fortran dummy arguments, or two C formal parameters. If both variables are dummy arguments or formal parameters, they must be arguments or parameters to the same procedure.

The following example shows how to specify COMMON-block variables, arguments, and parameters:

```
no_alias(PROCEDURE:DUMMY, /COMMONBLOCK/VAR)

no_alias(PROCEDURE:DUMMY1, PROCEDURE:DUMMY2)

no_alias(procedure:formal1, procedure:formal2)
```

The following example tells the Application Compiler that arguments `var1` and `var2` to a C function `foo` are not aliased:

```
no_alias(foo:var1, foo:var2)
```

A similar example for a Fortran routine looks like this:

```
no_alias(SUB1:A, SUB1:B)
```

The following statement tells the Application Compiler that a Fortran dummy argument and a COMMON-block variable are not aliased:

```
no_alias(SUB2:C, /COMMON1/D)
```

To specify a no_alias statement about a variable in an unnamed COMMON block, use two slashes, like this:

```
no_alias(SUB3:D, //E)
```

You can also use no_alias with the name of a single argument, which is the name of a COMMON block:

```
no_alias(/COMMONBLOCK/)
```

This tells the Application Compiler that the members of the COMMON block are not aliased to any dummy parameter anywhere in the program.

The following example shows how to use no_alias to make the Application Compiler ignore an alias. The program shown here has two apparent aliasing problems. In BAR, the dummy arguments C and D appear to be aliases. Also, in MOO, the dummy argument C and COMMON-block variable A appear to be aliases.

```
PROGRAM MAIN
REAL A(200)
COMMON A
CALL BAR(A(1), A(50))
CALL MOO(A(100))
END

SUBROUTINE BAR(C, D)
REAL C(50)
REAL D(50)
DO I=1, 50
  C(I) = C(I) + D(I)
  D(I+50) = C(I) - I
ENDDO
END

SUBROUTINE MOO(C)
REAL C(100), A(100)
COMMON A

DO I=1, 50
  C(I) = C(I) + A(I+50)
  A(I+50) = C(I) - I
ENDDO
END
```

Buildfiles

If it appears that there is an alias between two variables but the variables in question appear in a no_alias directive, the Application Compiler issues an advisory message that it is ignoring an alias because of a directive.

By examining the program, you can see that the code does not access overlapping elements of the apparently aliased arrays. Thus, a genuine alias does not exist. By adding the following lines to your buildfile, you can tell the Application Compiler to ignore the apparent aliases.

```
no_alias(//A, MOO:C)
no_alias(BAR:C, BAR:D)
```

The Application Compiler issues an advisory message to inform you that an apparent alias has been ignored.

Note

When a buildfile statement (inline, no_inline, clone, no_clone, or no_alias) uses a Fortran procedure or variable name, you must specify the name in uppercase only. Because standard Fortran does not distinguish between upper- and lowercase letters in names, Convex Fortran treats all names as uppercase. If your buildfile has Fortran procedure or variable names in mixed-case or lowercase, the Application Compiler cannot properly match the name. For C function names, you must use the same pattern of capitalization in your buildfile that you use in your source file. (Because C is case sensitive, the Application Compiler does not convert C function names.) Fortran COMMON-block names, COMMON-variable names, and dummy-argument names used with a no_alias statement also must be in uppercase.

var_args statement

The var_args statement in a buildfile performs the same function as a var_args directive in a source file: given a procedure that can have a variable number of arguments or parameters, it tells the Application Compiler how many arguments or parameters to consider significant for error checking. If the Application Compiler finds that a call does not pass enough arguments or parameters to a procedure, it issues a warning. Unless you tell it otherwise, the Application Compiler assumes that all arguments or parameters listed in a procedure declaration are required.

The following statement tells the Application compiler to consider only the first four arguments of procedure foo to be mandatory:

```
var_args(foo,4)
```

A call to foo that passes fewer than four arguments will cause a warning message; a call that passes four or more arguments will not.

Without the second argument, `var_args` tells the Application Compiler that all arguments are optional:

```
var_args(foo)
```

Specifying annotation mappings

If you choose to annotate libraries by compiling `.anno` files as described in Chapter 5, "Creating libraries and object files", these files are associated with the libraries they annotate by file name by default. For example, the annotation file `mylib.anno` would be associated with the library file `mylib.a` in the same directory.

If you would like to associate `.anno` files that are not stored in the same directory as or have different root names than the files they annotate, or if you would like to annotate all calls to a particular procedure using a particular annotation file, you can do so using the `annotate` buildfile statement. It has the following form:

`annotate` (*file_or_procname*, *anno_file*)

where *file_or_procname* is the name of the library file, object file, or procedure entry point name that you wish to annotate with the annotations contained in *anno_file*. Consider the following examples:

```
annotate(lib1.a, lib.anno)
annotate(test.o, test.anno)
annotate(compute, comput.anno)
```

The first statement tells the APC that the file `lib.anno` contains annotations for the library `lib1.a`. The second statement tells the APC that the file `testanno` contains annotations for the object file `test.o`. The third statement tells the APC that the file `comput.anno` contains the annotations for the procedure `compute`; regardless of where `compute` is defined, it is annotated according to `computanno`.

A single .anno file can be associated with multiple libraries or object files, as shown in the following example.

```
annotate(lib1.a, lib.anno)
annotate(lib2.a, lib.anno)
```

Here, both `lib1.a` and `lib2.a` use the annotations found in `lib.anno`.

Buildfiles

If you associate a single library or object file with multiple annotation files, as shown in the following example, the last annotation will override any that precede it.

```
#WARNING: following annotation is overridden:
annotate(lib1.a, libanno1)
#lib1.a is associated with libanno2 ONLY.
annotate(lib1.a, libanno2)
```

The `anno.map` file

The file `/usr/convex/apc/anno/anno.map` contains universal annotation mappings. This file is provided with the APC and is intended primarily to map Convex system libraries with their annotation files. It contains `annotate` statements which use syntax identical to that given above.

While most users will want to limit `annotate` entries to their buildfiles, you can edit `anno.map` to include mappings for system-wide libraries which you install yourself. Use caution when doing this as these mappings will be in effect for the specified libraries every time they are linked with an application.

`anno.map` annotation mappings are overridden by buildfile `annotate` statements; if the same file is mapped to different `.anno` files in `anno.map` and the buildfile, the buildfile mapping takes precedence.

For more information on annotation files, refer to Chapter 5.

Libraries and linking

To link libraries or precompiled object modules into your program, include the file name or path name of each library or module on the `link` line. For standard libraries, use `-l` followed by the library name. The following example links the Fortran program whose source is contained in `source.f` with the precompiled object file `obj.o`, a second object file `obj2.o`, and the `veclib` math library.

```
source.f -O2
link Fortran obj.o /mnt/me/objs/obj2.o -lveclib
```

Compiler options that affect library choices must appear on the `link` line as well as the compile line. These include the debugging option (`-db`), profiling options (`-p`, `-pdf`, and `-pg`), floating-point options (`-fi` and `-fn`), and language-compatibility options (such as `-cfc`, `-vfc`, or `-pcc`). For example,

```
foo.f -O2 -cfc
link Fortran -cfc
```

To rename an output file, you can use the -o option on the link line,

```
link Fortran -o foo
```

or use the -o option on the command line when you invoke apc:

```
% apc -o foo
```

To carry out interprocedural optimization, the Application Compiler requires information about all procedures, including library procedures. If this information is not available, optimization may be severely hampered. The Application Compiler derives this information about procedures from program source code, when it is available, or from special annotations that you can specify for libraries or object code for which you don't have the source. These annotations are provided for the Convex system libraries used by the APC.

If the source for an unannotated library or object file is not available, annotate the code using one of the procedures described in Chapter 5. If this is not possible, you can link the library just as you would an annotated library in your buildfile; however, note that using unannotated files in your program can result in a significant loss of optimization. Refer to Chapter 5 for information on annotating object files.

Linking indirectly called procedures

To reduce compilation time and produce smaller executables, the Application Compiler may not link procedures that are found in your source files but never called. Occasionally, this can cause problems. Some programs define procedures, such as VAX-binary I/O or conversion routines, that are not called directly but are used by runtime libraries. The Application Compiler sees these routines are not called and does not link them into the finished program. Likewise, routines called only by library routines appear to be dead code, and the Application Compiler does not link them. This is true even if the routine called by the unannotated procedure was compiled with the APC itself; if it is not also called by an APC-compiled procedure, it is considered dead code.

If the Application Compiler fails to link a procedure into the final executable and issues an "unresolved references" advisory at link time, use the force_object statement in your buildfile. This command forces the Application Compiler to include a procedure in your executable, even if that procedure appears to be uncalled.

For example, to force the Application Compiler to include procedure `PROC1` in your executable, use

```
force_object(PROC1)
```

in your buildfile. To force the inclusion of more than one procedure, include a line like this:

```
force_object(PROC1, PROC2, PROC3)
```

Keep in mind that if you get an "unresolved symbols" advisory and recompile after inserting a `force_object` statement into your buildfile, compilation proceeds from part one of the synthesis phase for the entire program.

Directories

To compile source files, you can specify as many file names as you like on multiple lines. For a large program with many source files, however, this can be inconvenient. An alternative is to specify the directory containing the source files to compile. The Application Compiler looks in the specified directory for source files and compiles them.

The buildfile in the following example compiles the source files found in `/mnt/me/zwork/prj/src` at optimization level `-O2`.

```
/mnt/me/zwork/prj/src -O2
link Fortran
```

If you keep your buildfile in the same directory as your source files, the syntax of your buildfiles can be even simpler. `apc` recognizes the standard UNIX dot notation for the current directory. The following buildfile causes the Application Compiler to compile the program whose source is found in the current directory.

```
. -O2
link C
```

You can specify default options for the files in a directory on one line and specify different options for specific files on other lines. The following example shows a buildfile that compiles `foo.f` at `-O3`, compiles the rest of the program in the current directory at `-O2`, and links the resulting object code with the standard Fortran library.

```
. -O2
foo.f -O3
link Fortran
```

Note

If you specify directories rather than file names, beware of any extraneous C or Fortran files that may be in those directories. Such files can cause confusion if the Application Compiler finds duplicate procedure names. In particular, beware of unused files that contain `main()` functions or program `MAIN` sections. If the Application Compiler finds more than one main entry point to a program, it cannot determine which one to use and therefore cannot produce an executable. It generates the error message shown in Figure 41.

```
Error: More than one 'main' entry point found in ./myfile.f
```

Figure 41 Multiple main error message

This condition can occur because of the back-up files that some editors leave in a directory. When you edit `myfile.f`, for example, the `edt` editor creates a back-up file named `.#myfile.f`. If you expect to have multiple copies of files containing your `MAIN` procedure in the same directory, specify the exact file names in your buildfiles instead of using the directory shorthand.

Flag macros and options statements

You can define two flag macros, FFLAGS and CFLAGS, to represent default compilation options in your buildfile. The FFLAGS macro specifies default options for compiling Fortran; the CFLAGS macro specifies defaults for compiling C. The following example shows how these flags are defined and used.

```
FFLAGS = -re -rl -db
foo.f -O1
bar.f -O2
baz.f -O2
```

The Application Compiler concatenates the options specified in FFLAGS onto the end of the options list specified for each individual Fortran source file. File foo.f is therefore compiled with -O1 -re -rl -db, while bar.f and baz.f are compiled with -O2 -re -rl -db.

If apc is invoked from a makefile, the CFLAGS and FFLAGS macros can be set in the makefile instead. Consider the following buildfile:

```
main.f
marks.f -re
finn.f
mat.f -re
```

The corresponding makefile looks like this:

```
FFLAGS = -O2
tmp:
     apc -check types $(MFLAGS) "FFLAGS =
'$(FFLAGS)'"
```

The apc command executed in this makefile builds the above buildfile using the FFLAGS definition specified in the makefile.

Note the $(MFLAGS) macro used on the apc command line. This macro passes the -n, -i, and -s flags from the make command line through to apc. MFLAGS must contain only options supported by apc.

If your application has source files spread out over several directories, and one or more directories contain include files, use FFLAGS in your buildfile to pass the proper directory search path to the Application Compiler. Figure 42 shows a directory containing source files to be included in a buildfile.

```
% ls -l /mnt/me/applic
drwxrwx---   2 me    1536 Jul  3 15:03 include
drwxrwx---   2 me    512 Jul  3 14:58 src.a-f
drwxrwx---   2 me    512 Jul  3 14:58 src.g-l
drwxrwx---   2 me    512 Jul  3 14:58 src.m-r
drwxrwx---   2 me    512 Jul  3 14:58 src.s-z
```

Figure 42 Example source file organization

Figure 43 shows the buildfile for the source files of Figure 42.

```
FFLAGS = -I/mnt/me/applic/include
./src.a-f -O2
./src.g-l -O2
./src.m-r -O2
./src.s-z -O2
options -v
link Fortran
```

Figure 43 Buildfile for sources of Figure 42

If you do not pass the `-I` option with `FFLAGS`, you must include it as an option for each of the compile statements in the buildfile, as shown in Figure 44.

```
./src.a-f -O2 -I/mnt/me/applic/include
./src.g-l -O2 -I/mnt/me/applic/include
./src.m-r -O2 -I/mnt/me/applic/include
./src.s-z -O2 -I/mnt/me/applic/include
options -v
link Fortran
```

Figure 44 Buildfile excluding `FFLAGS` statement

The keyword `options` allows you to specify command line options in your buildfile instead of on the command line. Invoking `apc` without any command line options and using the buildfile shown in Figure 45 produces the same results as invoking `apc` with `-check types` and `-show all` on the command line. If you specify additional options on the command line, `apc` uses those as well.

```
options -check types -show all
. -O2
foo.f -O3
bar.c -O1
link Fortran
```

Figure 45 Buildfile with `options` keyword

If you specify a command line option that conflicts with an option in a buildfile `options` statement, the command line option either overrides the buildfile option or generates an error and stops compilation. The options shown in Table 10, when used on the `apc` command line, override conflicting options found in the buildfile.

Table 10 Overridable `options` statement options

Option	Overrides
-inline low	–inline medium –inline high
-inline medium	–inline low –inline high
-inline high	–inline low –inline medium
-clone none	–clone all
-clone all	–clone none
-s	–v
-v	–s

ANSI Fortran standard checking

The Application Compiler provides `lint`-like checking for ANSI Fortran standard violations. This checking can detect many, though not all, violations of the ANSI 77 and ANSI 90 Fortran standards. Many Fortran programs, especially older codes, do not conform to these standards and generate a large number of warnings when these checks are performed. For this reason, ANSI Fortran standard checking is not performed by default. To instruct the Application Compiler to perform ANSI error checking, you must use the following compiler option:

```
-chk
```

You must also use at least one of these two options:

```
-ansi77
-ansi90
```

Do not place these options on the `apc` command line. Place them on the source line within the buildfile instead. For example,

```
. -O2 -chk -ansi77
link Fortran
```

This tells the Application Compiler to perform ANSI 77 error checking on all files in your current directory when it compiles them.

If you do not use these compiler options, the Application Compiler does not perform the ANSI error checks and does not store ANSI messages in the program database. If you view the program database with the PDB Viewer, you will not see ANSI messages unless the program was compiled with these options.

Visual debugging

Convex CXdb is an optional window-based debugger that includes all the functionality of regular debuggers and is capable of debugging Application Compiler generated code. Cxdb can debug any Convex Fortran or C executable; however, to fully employ the power of CXdb, the program should be compiled with the `-cxdb` option. Failure to compile with `-cxdb` will prevent access to symbolic debugging information. Like any other APC-supported compiler option, you can supply the `-cxdb` option on the source file compilation line in your buildfile. The file source.f in Figure 40 is compiled for debugging with CXdb.

Refer to the *CXdb User's Guide* for more information.

Combining C and Fortran

Writing a buildfile that combines C and Fortran procedures is almost as easy as writing a buildfile for a single language. The following example shows a buildfile for a program written in C and Fortran:

```
foo.c
bar.c
baz.c -O2
main.f -O2

link Fortran
```

If you are compiling a program that uses both languages, write the main routine in Fortran and use `Fortran` on the `link` line, as shown here. If, for some reason, the main routine must be written in C, you must use `C` on the link line and specifically add the Fortran libraries that your program needs using the `-l` option. For example, if your program needs the `F77` library (in addition to the standard C libraries), the `link` line looks like this:

```
link C -lF77
```

When calling procedures from another language, make sure that arguments and calling method (call by reference, call by value) match. Consult the *Fortran Guide* or *C User's Guide* and the *Interlanguage Programming Guide* for information on how to make your calls to another language compatible.

Because of semantic differences between Fortran and C, the Application Compiler cannot inline Fortran procedures into C, or C procedures into Fortran. Mixing languages, therefore, reduces the amount of inlining the Application Compiler performs.

Creating libraries and object files

5

Effective interprocedural optimization depends on the Application Compiler having information on the behavior of all the procedures in a program. Without this information, the Application Compiler must make worst-case assumptions that can severely inhibit optimization.

When you use `apc` to compile source code, the Application Compiler automatically gathers the information it needs from the source files and stores it in an intermediate representation (IR) format in the program database. This allows full interprocedural optimization of the procedures in question.

Unfortunately, the Application Compiler has no information on procedures that you do not compile yourself, such as those contained in libraries. Two mechanisms are available to you to allow the APC to use such code more effectively.

The easiest and most beneficial method involves using `apc` to compile your sources into IR-format files known as APC libraries. Procedures contained in these libraries are optimized to the same extent as the procedures in your program. APC libraries are discussed in detail in the next section.

The other method requires you to specify *annotations* for use with your object file. These annotations consist of procedure summary (psum) directives and pragmas that are used to explain the behavior of your source, and are stored in a separate file, called a *stub* file. While annotated object files cannot be modified by the APC and therefore cannot be optimized to the same extent as APC libraries, they nevertheless allow more complete optimization of the procedures that use them.

Psum directives are designed to be specified through use of a stub file, as described further on. The standard Convex libraries used by the APC employ these annotations.

Building and using APC libraries

An APC library is a special IR-format library for use with the Application Compiler. These libraries are larger than standard libraries, but they allow full interprocedural optimization with any procedures they are linked with, and, because they are precompiled, they reduce compile time. If you have access to library source, you can create your own APC libraries using `apc` with the `-library` option.

To generate an APC library, first create a buildfile just as you would to compile a non-library. Link lines are not required when using the `-library` option since the APC library generated will be linked with other procedures later; link lines are therefore ignored when the `-library` option is used.

A simple buildfile for building an APC library might look like this:

```
# buildfile for myarchv
# apc with -library option only
libsource1.c -pa -O2
libsource2.c -pa -O2
oldlib.c -pa -pcc -O2
```

The apc command line to compile this file and name it "`myarchv.apclib`" is shown below:

```
% apc -library -o myarchv
```

When this line is executed, `apc` performs the summary phase and certain portions of the synthesis phase, generating an IR-format file called `myarchv.apclib` in the process. The remaining phases of compilation are completed when this library is included in a buildfile.

Once you build an APC library, you can use it in a build by including an `apc_libraries` statement in the executable's buildfile. For example, if you were building an executable made up of the procedures `proc1.c` and `proc2.c` and the APC libraries `lib1.apclib` and `lib2.apclib`, the buildfile might look like this:

```
proc1.c -O2 -pa
proc2.c -O2 -pa
apc_libraries lib1.apclib lib2.apclib
link C -pa
```

The file names following the `apc_libraries` statement tell the Application Compiler which APC libraries to use. The order in which libraries are specified on this line depends on how the libraries call one another. Procedures in one library can call other procedures in the same library. They can also call procedures in any library that appears to the right of the calling library on the

apc_libraries line. They cannot call procedures in any library that appears to the left of the calling library. So the following example, where the useslib1.apclib library contains procedures that use lib1.apclib, is incorrect:

```
# INCORRECT:
  apc_libraries lib1.apclib useslib1.apclib
```

The correct form of the apc_libraries line is shown below:

```
# CORRECT:
  apc_libraries useslib1.apclib lib1.apclib
```

Using psum directives

Procedure summary (psum) directives are the special compiler directives and pragmas used to summarize the behavior of library or object file routines for which the Application Compiler does not have the source. While annotated procedures cannot themselves be optimized at compile time, the annotations associated with them help the APC to optimize the procedures in your program that call them. Annotations are placed into a stub file, which is "compiled" using the appropriate procedural compiler and associated with the appropriate object or library file.

Stub file format

A stub file contains a series of procedure summaries, which describe interface and side effect information about the procedures contained in object files or libraries. Procedure summaries can be written in either Fortran or C, regardless of what language the procedure being described is written in; since the source is not available, its language is irrelevant.

A procedure summary must contain procedure header information, variable declarations, and a terminal statement; it may contain executable code, but if it does, the code is ignored by the compiler. A procedure summary written in Fortran, for example, has this format:

```
<subroutine or function header>
<argument declarations>
<psum directives>
END
```

A procedure summary written in C follows this format:

```
<function header>
{
 <psum pragmas>
}
```

Note that by definition a C function header includes argument declarations.

Write the declarations and procedure headers exactly the way you would in a Fortran or C source file. Make sure the argument declarations in your stub files match those in your calling-procedure source files.

If a variable name appears in any psum directive, make sure that a variable declaration for that name also appears in the same procedure as the directive (or in the same sourcefile for a global variable). The Application Compiler requires these declarations to perform proper type checking.

A simple stub file

The following example shows a stub file written in Fortran for the Convex MLIB procedures SAXPY and SDOT. Library products such as MLIB are generally provided precompiled into object code, with no source code available, but the documentation provides a Fortran equivalent to the procedure which shows argument declarations and the general algorithm. This information was used to determine which psum directives are needed. In this file, they tell the Application Compiler that neither SAXPY nor SDOT perform I/O, that both use all of their arguments, and that SAXPY assigns a value to the argument Y. (The next section explains these and other psum directives in detail.)

```
      SUBROUTINE SAXPY(N,A,X,INCX,Y,INCY)
      REAL*4 A
      REAL*4 X(*), Y(*)
      INTEGER*4 N, INCX, INCY
C$DIR PSUM_USES(A,X,N,INCX,Y,INCY)
C$DIR PSUM_ASGS(Y)
C$DIR PSUM_NO_IO
      END

      REAL*4 FUNCTION SDOT(N,X,INCX,Y,INCY)
      REAL*4 X(*), Y(*)
      INTEGER*4 N, INCX, INCY
C$DIR PSUM_USES(N,X,INCX,Y,INCY)
C$DIR PSUM_NO_IO
      SDOT=1.0
      END
```

Note

These Convex MLIB routines are used for illustrative purposes only. The Convex MLIB libraries are shipped with `.anno` files; you do not have to create them yourself.

An assignment to SDOT is included in this stub file because the Fortran compiler will issue a warning if it is omitted. This has no effect on the generation of the .anno file, which is correctly generated whether the assignment is included or not. Including it simply prevents the warning message from being issued.

The next example shows the same stub file written in C, with psum directives inserted as C pragmas.

```
void saxpy(n, a, x, incx, y, incy)
float a;
float x[], y[];
int n, incx, incy;
{
#pragma _CNX psum_uses(n,a,*x,incx,*y,incy)
#pragma _CNX psum_asgs(*y)
#pragma _CNX psum_no_io
}

float sdot(n, x, incx, y, incy)
float x[], y[];
int n, incx, incy
{
 #pragma _CNX psum_uses(n,*x,incx,*y,incy)
 #pragma _CNX psum_no_io
}
```

Compiling stub files

Compile stub files using the -anno option with cc if the stub file uses C syntax, or fc if the stub file uses Fortran syntax. Which compiler to use depends only on the language of the stub file; the language of the object or library file is irrelevant, since you don't have access to the source.

For example, to compile the Fortran-language stub file lib1.f, you would use the following command line:

```
fc -anno lib1.f
```

Compilation produces an *annotation* file containing annotation information in an abbreviated form that the APC can understand. This text file has the same root name as the original, but with a .anno suffix; in the above example, the compiled annotation file has the name lib1.anno. By default, the .anno file is associated with the file it annotates via its file name root, and must be in the same directory as the target file. In the preceding example, the annotation file lib1.anno would be associated with the library lib1.a or the object file lib1.o by default, if one of these files

existed in the same directory as `lib1.anno`. This behavior can be overridden by specifying an `annotate()` statement in the buildfile that includes the annotated library, as discussed in Chapter 4.

Psum directives

The following directives can be used in stub files to provide the APC with information about the behavior of procedures for which you do not have access to the source code.

- `PSUM_ASGS`
- `PSUM_KILLS`
- `PSUM_USES`
- `psum_range_names` (C only)
- `psum_range_flags` (C only)
- `psum_args_dealloc` (C only)
- `PSUM_REENTRANT`
- `PSUM_VAR_ARGS`
- `PSUM_NO_IO`

Note

Note that psum directives are unrelated to the compiler directives described in Chapter 6, "Directives". Psum directives should only be used in stub files; the APC ignores psum directives that appear in the source code it compiles.

The following sections describe each psum directive in detail. The C formats given use pragma notation, since directive notation in C is being phased out.

When writing a C pragma, keep in mind that C passes variables to functions by value only. Whenever an array is used as a function argument, it is actually the address of the array that is passed to the function. Therefore, when specifying array arguments in psum pragmas in C, you must indicate that the argument is a pointer by using pointer notation; that is, you must precede the argument with an asterisk (*). Explicitly declared pointers must also be specified in this manner.

Note

When you use these directives to create stub files, be sure to consider variables that appear farther down the call tree than the current procedure, and add appropriate annotations and declarations for them if necessary. If procedure `PROC1`, for example, always calls procedure `PROC2`, which may assign variable `A`, the stub file for `PROC1` must include a `PSUM_ASGS` directive for `A`, even if that variable is never assigned in the body of procedure `PROC1`. To prevent type checking errors, `A` must also be declared in `PROC1`.

PSUM_ASGS

The PSUM_ASGS directive tells the Application Compiler that the procedure, if invoked, may assign a value to a certain variable (either a global or an argument), depending on what happens at runtime. As shown here, the PSUM_ASGS directive takes the name of a single variable or a list of variable names separated by commas:

C format
#pragma _CNX psum_asgs(a)
#pragma _CNX psum_asgs(a,b,*c)

Fortran format
C$DIR PSUM_ASGS(A)
C$DIR PSUM_ASGS(A,B,C)

In the final C example, the variable c is a pointer or array variable.

PSUM_KILLS

The PSUM_KILLS directive makes a stronger statement than the PSUM_ASGS directive. PSUM_KILLS tells the Application Compiler that the procedure, when invoked, *always* assigns a value to a global variable or argument, *regardless* of what happens at runtime. The PSUM_KILLS directive takes the name of a single variable or a list of variable names separated by commas:

C format
#pragma _CNX psum_kills(a)
#pragma _CNX psum_kills(a,b,*c)

Fortran format
C$DIR PSUM_KILLS(A)
C$DIR PSUM_KILLS(A,B,C)

In the final C example, the variable c is a pointer or array variable.

Note

Using the PSUM_KILLS directive on a variable that is not assigned can result in wrong answers; when in doubt, use PSUM_ASGS.

PSUM_USES

The PSUM_USES directive is similar to the PSUM_ASGS directive in that it states what *may* happen at runtime. In this case, the directive tells the Application Compiler that the procedure, when invoked, may reference the value of a global variable or argument. As with PSUM_ASGS and PSUM_KILLS, the directive takes the name of a single variable or a list of variable names separated by commas:

C format
#pragma _CNX psum_uses(a)
#pragma _CNX psum_uses(a,b,*c)

Fortran format
C$DIR PSUM_USES(A)
C$DIR PSUM_USES(A,B,C)

In the final C example, the variable c is a pointer or array variable.

---

psum_range_names

Use the psum_range_names directive on a C function that returns a pointer. The argument is a variable name or list of variable names, separated by commas, that tell the Application Compiler what data objects the return pointer can point to.

The first example shown below indicates that the procedure returns a pointer to a. The second example shows that the pointer can point to a, b, c, or f.

C format
#pragma _CNX psum_range_names(a)

C format
#pragma _CNX psum_range_names(a,b,c,f)

psum_range_flags

Use the psum_range_flags directive with a C function that returns a pointer to memory allocated on the heap. For example, assume you have a function valloc that uses malloc to allocate a block of memory returned by valloc:

```
char *valloc()
{
 int n;
 char *p;
 ...
 p= malloc(n);
 ...
 return p;
}
```

Using the range flags directive as shown below tells the Application Compiler that valloc returns a pointer to a unique object on the heap:

C format
#pragma _CNX psum_range_flags(valloc, heap)

You do not need to use the psum_range_flags directive on the standard malloc function, which is already annotated. However, if you write your own malloc replacement, it should be annotated with psum_range_flags to aid optimization.

psum_args_dealloc

Use the psum_args_dealloc directive for a C function that deallocates storage space for one or more arguments within the scope of the procedure. If a procedure deallocates storage for the argument arg_name, for example, write the psum directive as follows:

C format
#pragma _CNX psum_args_dealloc(arg_name)

PSUM_REENTRANT

The PSUM_REENTRANT directive tells the Application Compiler that the procedure has been coded for reentrancy. Reentrant procedures can be safely called from a parallel loop, whereas non-reentrant procedures cannot. The directive format is as follows

C format
`#pragma _CNX psum_reentrant`

Fortran format
`C$DIR PSUM_REENTRANT`

PSUM_VAR_ARGS

The PSUM_VAR_ARGS directive in a stub file performs the same function as the VAR_ARGS directive in a source file or a var_args statement in a buildfile: given a procedure with a variable number of arguments or parameters, it tells the Application Compiler how many arguments or parameters to consider significant for error checking. If the Application Compiler finds that a call does not pass enough arguments or parameters to a procedure, it issues a warning. Unless you tell it otherwise, the Application Compiler assumes that all arguments or parameters listed in a procedure declaration are required.

The PSUM_VAR_ARGS directive tells the Application Compiler how many arguments to consider mandatory. The following use of PSUM_VAR_ARGS tells the Application Compiler that only the first two arguments are required.

C format
`#pragma _CNX psum_var_args(2)`

Fortran format
`C$DIR PSUM_VAR_ARGS(2)`

Without an argument, PSUM_VAR_ARGS tells the Application Compiler that all arguments are optional:

C format
`#pragma _CNX psum_var_args`

Fortran format
`C$DIR PSUM_VAR_ARGS`

PSUM_NO_IO

Certain optimizations, such as code motion and parallelization, can be dangerous when performed on calls to procedures that do I/O; therefore, unless a procedure is otherwise annotated, the APC assumes that it does I/O. The PSUM_NO_IO directive indicates that a procedure and its children do not perform I/O, allowing these optimizations to be performed.

Fortran format	C format
`C$DIR PSUM_NO_IO`	`#pragma _CNX psum_no_io`

Note

I/O operations include any operations that affect the state of the operating system, not just reading and writing. Use this directive with caution.

Example annotation

Assume you have an object file called dea.o which contains a procedure called trample(). From the documentation, you know that trample has the following interface:

```
int trample(int amd4, int amd5, char *pdfa)
```

The documentation also tells you that trample is written in ANSI C, that it always assigns a value to its arguments amd4 and amd5, it may assign to the global variable morls, and that it may reference the string pointed to by pdfa.

Since you only have `trample`'s object code, you must create a stub file describing it and compile the file using `cc -anno`. From the information given above, you can construct a stub file, called `dea.c`, as shown below.

```
int morls;
int trample(int *amd4,int *amd5,char *pdfa)
{
 #pragma _CNX psum_kills(*amd4, *amd5)
 #pragma _CNX psum_uses(*pdfa)
 #pragma _CNX psum_asgs(morls)
}
```

Now you must compile `dea.c` with `cc -anno` to create `dea.anno`, which is the annotation file that will be used by the APC. Your `cc` command line will look like this:

```
cc -anno -ext dea.c
```

It is necessary to use the `-ext` option so that ANSI mode is used during compilation.

This will create the file `dea.anno`, which will, by default, be associated with `dea.o`, assuming `dea.o` exists in the same directory.

If you want to store `dea.o` in a different directory, you can add an `annotate` statement to the buildfile used to link `dea.o` with the other files in the application. An example buildfile follows.

```
. -O3
annotate(/work/libs/dea.o,/work/anno/dea.anno)
link C /work/libs/dea.o
```

This compiles all modules in the current directory at optimization level `-O3`, annotates the object file `/work/libs/dea.o` using the annotations contained in `/work/anno/dea.anno`, and links the entire program with the annotated `dea.o`.

anno_ar utility

Compiled .anno files can be manipulated using the anno_ar(1) utility. The anno_ar utility has the following command line:

anno_ar *option anno_file* [...] [*entry_point*]

where *anno_file* is a .anno file, *entry_point* is a space-delimited list of one or more entry points, and *option* is one of the following options:

-h	Print a help message.
-i	Create an index for one or more *anno_files*. An index is a header section of the compiled annotation file that lists the routines defined in the file. -i can be specified with other options that require an index.
-t	List the entry points described in one or more *anno_files*.
-r	Replace annotations for entry points in the first *anno_file* with annotations for the same entry points in the second *anno_file*; if the second *anno_file* contains entry points not present in the first, they are appended. More than two *anno_files* can be specified, in which case annotations in each successive file replace or are appended to annotations in the first file.
-x	Extract the annotations for the entry points specified in *entry_point* from *anno_file* and print them to the standard output.
-v	Produces verbose listing when used with -f, -t, or -x.
-d	Delete the entry points listed in *entry_point* from *anno_file*.

When specified with other options, the -i option will cause indexes to be created only for the files that need them.

Examples

Assume you have a library called `liblocal.a`, and it contains three procedures: `check1`, `check2` and `check3`. From what you know about how these procedures work, you construct a Fortran stub file called `liblocal.f`, which is shown below.

```
      SUBROUTINE CHECK1(I, J, K)
      REAL*16 J
C$DIR PSUM_USES(I, J)
C$DIR PSUM_ASGS(K)
      END

      SUBROUTINE CHECK2(A, B)
      COMMON /BLK1/ X, Y, Z(10)
C$DIR PSUM_USES(A, X, Y)
C$DIR PSUM_USES(B)
C$DIR PSUM_ASGS(B, Y)
      END

      REAL*4 FUNCTION CHECK3(T)
C$DIR PSUM_USES(T)
      END
```

Creating and viewing an index

Compiling `liblocal.f` with `fc -anno` produces the annotation file `liblocal.anno`. This readable text file contains all the annotation information derivable from `liblocal.f`, but in an abbreviated format for the Application Compiler. To create an index for `liblocal.anno`, use `anno_ar -i`, as shown:

```
anno_ar -i liblocal.anno
```

Now you can use other `anno_ar` options to manipulate `liblocal.anno`. For example, to view the entry points it contains, use the -t option:

```
anno_ar -t liblocal.anno
```

which produces:

C Series only

```
_check1_
_check2_
_check3_
```

On C Series machines, and:

SPP Series only

```
check1
check2
check3
```

on SPP Series machines.

Extracting entry point information

Use the appropriate given entry point names with other options that require them, such as -x. The following example extracts information for the procedure check3 on a C Series platform:

```
anno_ar -x liblocal.anno _check3_
```

This yields:

```
entry_name _check3_
uses T
arguments T
base_types
notype:int1:logical1:logical2:logical4:logical8
:int2:int4:int8:real4:real8:real16:complex8:uns
igned1:unsigned2:complex16:unsigned4:unsigned8:
char:hollerith:bitpattern:void_type:int16:schar
:long:long_double:addr_exp:ulong:hollerithzf
type 29 function 9
language fortran
symbol CHECK3 entry_type=variable
sclass=static type=9
symbol T entry_type=variable sclass=arg_ptr
type=9
```

This is the compiled annotation information for the procedure check3. It contains both information from user-supplied psum directives ("uses T") and information that the APC constructed itself. This information is stored as readable text in the file liblocal.anno.

Note

While .anno files are readable text, manually editing them is strongly discouraged.

Replacing/appending annotations

The -r option allows you to concatenate two or more annotation files, with entry point annotations contained in initial files being overwritten by annotations for the same entry point contained in later files.

For example, assume we have two new annotation files: libx.anno and liby.anno.

```
anno_ar -t libx.anno
```

run on a C Series machine produces:

C Series only

```
_check2_
_check4_
_test1_
_test2_
```

(on an SPP Series machine, the underscores would not be present) and

```
anno_ar -t liby.anno
```

run on a C Series machine produces:

C Series only

```
_check1_
_check2_
_test2_
_test3_
```

Executing the following command:

```
anno_ar -r liblocal.anno libx.anno liby.anno
```

Causes annotations from liby.anno to be merged with those in libx.anno, then causes the resulting annotations to be merged with liblocal.anno. Both libx.anno and liby.anno are not modified.

Checking the index of liblocal.anno with anno_ar -t now yields:

C Series only

```
_check1_
_check4_
_check2_
_check3_
_test2_
_test1_
_test3_
```

Only check3 still has its original annotations from liblocal.anno; check1 and check2 now contain annotations from liby.anno. test1 still has its annotations from libx.anno, but test2 has annotations from liby.anno. test3 was appended from liby.anno.

Directives

6

The most significant advance to occur with the introduction of the Application Compiler is the addition of interprocedural optimizations. Because of the type of information the Application Compiler generates, these optimizations are safer and more efficient than the optimizations that the conventional compilers perform. In some cases, however, you may want to override the Application Compiler's decisions on certain optimization issues or augment those decisions with additional information. The Application Compiler, therefore, adds several new directives to allow you to control interprocedural optimization. These directives let you do the following:

- Control the automatic inlining of procedures
- Control the cloning of procedures
- Provide additional information about conditional statements and loops

The following sections discuss these three types of directives.

Inlining directives

Inlining directives prevent or force inlining of specific procedures or calls. Adding directives to a program can be time consuming, so if your goal is simply to increase or decrease the number of calls inlined, use the `-inline` option described in Chapter 3. The inlining directives are

- `INLINE`
- `NO_INLINE`
- `INLINE_CALL`
- `NO_INLINE_CALL`

INLINE

Ordinarily, the Application Compiler does not inline infrequent calls or calls to lengthy procedures. The `INLINE` directive

overrides the Application Compiler's profitability analysis and forces inlining of specified calls. If you believe that a procedure is called more often than the compiler realizes, use this directive to inline the calls.

The `INLINE` directive is placed just before the body of a procedure (after the subroutine or function declaration). In C, it must appear inside the brackets of the function body. If it appears elsewhere, the Application Compiler issues a warning message and ignores the directive.

The following example shows how the `INLINE` directive is used with C and Fortran functions.

Fortran code	**C code**
`PROGRAM MAIN`	`main{`
	`float a[1000], b;`
`REAL A(1000), B`	`int j;`
`INTEGER J`	`b=ros(a[1]);`
`B=ROS(A(1))`	`for (j=0; j<100; ++j)`
`DO J=1,100`	`  a[j]=ros(a[j]);`
`  A(J)=ROS(A(J))`	`}`
`ENDDO`	
`END`	`float ros(x)`
	`float x;`
`FUNCTION ROS(X)`	`{`
`C$DIR INLINE`	` int i;`
`REAL X, P`	` float p;`
`INTEGER I`	` #pragma _CNX inline`
`P=0.0`	` p=0.0;`
`DO I=1,200,2`	` for (i=0; i<200; i+=2)`
`  P=P+SIN(X/(I+1))`	`   p+=sin(x/(i+1));`
`ENDDO`	` return(p/x);`
`ROS=P/X`	`}`
`END`	

Using the `INLINE` directive in this manner causes the Application Compiler to inline both calls to `ROS`, producing code equivalent to this:

Fortran code	C code
`P=0.0`	`p=0.0;`
`DO I= 1,200,2`	`for (i=0; i<200; i+=2)`
`  P=P+SIN(A(1)/(I+1))`	`  p=sin(a[1]/(i+1));`
`ENDDO`	
`B=P/A(1)`	`b=p/a[1];`
`DO J=1,100`	`for (j=0; j<100; ++j){`
`  P=0.0`	`  p=0.0;`
`  DO I=1,200,2`	`  for (i=0;i<200;i+=2)`
`    P=P+SIN(A(J)/(I+1))`	`    p+=sin(a[j]/(i+1));`
`  ENDDO`	`  a[j]=p/a[j];`
`  A(J)=P/A(J)`	`}`
`ENDDO`	

NO_INLINE

The NO_INLINE directive has the opposite function of INLINE; it prevents the Application Compiler from inlining any call to a specified procedure, no matter how small the procedure is or how often the call is made. NO_INLINE is useful when:

- you find that the Application Compiler inlines a procedure that is seldom called.
- you find the inlining of a particular procedure causes a significant increase in your program's size, and you do not believe the increased speed due to inlining justifies the additional memory usage. This is especially true if you notice an increase in memory swapping when you execute your program after compiling with the Application Compiler.
- the procedure contains no inhibitors of parallelization, is called from a parallelizable loop, and you suspect that performance would benefit if the loop was parallelized with the call intact rather than inlined. See Chapter 2, "Basic interprocedural optimizations", for more information on parallelizing loops with calls.

The NO_INLINE directive is placed just before the procedure body. If it appears elsewhere, the Application Compiler generates a warning message and ignores the directive.

The following example shows how NO_INLINE is used to prevent the inlining of a C or Fortran function.

Fortran code	C code

`      FUNCTION ROS(X)`
`C$DIR NO_INLINE`
`      REAL X, P`
`      INTEGER I`
`      P=0.0`
`      DO I= 1,200,2`
`        P=P+SIN(X/(I+1))`
`      ENDDO`
`      ROS=P/X`
`      END`
	`ros(x)`
	`float x;`
	`{`
	` int i;`
	`#pragma _CNX no_inline`
	` p=0.0;`
	` for (i=0;i<200;i++)`
	`   p+=sin(x/(i+1));`
	` return(p/x);`
	`}`

INLINE_CALL

If you believe that a single call to a procedure is made more often than the compiler realizes, you can force the Application Compiler to inline that particular call. To do this, use the INLINE_CALL directive. This directive appears before a statement in the source code containing one or more calls. It can be followed by a set of parentheses containing a list of procedures that are called in the next statement. Without the parentheses and procedure list, INLINE_CALL forces the Application Compiler to inline all calls made by the next statement. In the following code, for example, the INLINE_CALL directive causes the Application Compiler to inline both calls.

Fortran code

```
C$DIR INLINE_CALL
      A=ROS(GMA)-SOR(GMA)
```

C code

```
 #pragma _CNX inline_call
  a=ros(gma)-sor(gma);
```

If you add a procedure list to the directive, INLINE_CALL no longer forces the Application Compiler to inline every call in the following statement. Instead, it forces the compiler to inline calls to those procedures named in the list.

The following code shows how INLINE_CALL is used with a procedure list. The first use of the INLINE_CALL directive causes the Application Compiler to inline both calls to ROS in the first statement. The call to SOR may or may not be inlined, depending on the Application Compiler's profitability analysis.

Fortran code

C$DIR INLINE_CALL(ROS)
      A=ROS(GM)*ROS(DL)-SOR(GM)
C$DIR INLINE_CALL(ROS, FN1)
      B=ROS(DL)/FN(THETA)-FN1(B)

C code

#pragma _CNX inline_call(ros)
   a=ros(gm)*ros(dl)-sor(gm);
#pragma _CNX inline_call(ros, fn1)
   b=ros(dl)/fn(theta)-fn1(b);

The second INLINE_CALL directive causes the Application Compiler to inline the calls to ROS and FN1 in the second statement. The Application Compiler inlines the call to FN only if its analysis shows that it is profitable to do so.

An INLINE_CALL directive has priority over a NO_INLINE directive. If you use an INLINE_CALL directive on a call to a procedure that has a NO_INLINE directive, the Application Compiler inlines that specific call. In the following code, for example, the INLINE_CALL directive causes the Application Compiler to inline the first call to ROS despite the NO_INLINE directive. The second call to ROS is not inlined.

Fortran code	C code
PROGRAM MAIN	main{
	float a[1000], b;
REAL A(1000), B	int j;
INTEGER J	#pragma _CNX inline_call
C$DIR INLINE_CALL	b=ros(a(1));
B=ROS(A(1))	for (j=0; j<100; ++j)
DO J=1,100	a[j]=ros(a[j]);
A(J)=ROS(A(J))	}
ENDDO	
END	ros(x) {
FUNCTION ROS(X)	#pragma _CNX no_inline
C$DIR NO_INLINE	/* function body
C function body	omitted for clarity */
C omitted for clarity	}
END	

Note that none of the inlining directives can be used on an indirect call. In the following example, a pointer to the function bar is passed to function foo, which then calls bar indirectly. The

Application Compiler cannot inline an indirect call, so it ignores the INLINE_CALL directive and generates a warning message.

```
typedef int (*funcPtr)();
int bar();

main(){
 ...
 foo(bar);

}

foo(f)
funcPtr f;
{
 /* bad directive use: */
#pragma _CNX inline_call
(*f)();
}
```

NO_INLINE_CALL

If you believe that a particular call to a procedure is seldom made, but the Application Compiler does not realize that, you may want to use the NO_INLINE_CALL directive to prevent inlining of that call. Also, if the inlining of a particular call causes a significant increase in your program's size, and you do not believe the increased speed due to inlining justifies the additional memory usage, consider using this directive. This is especially true if you notice an increase in memory swapping when you execute your program after compiling with the Application Compiler.

Like the INLINE_CALL directive, the NO_INLINE_CALL directive appears immediately before the source line of a call. If no calls appear on the source line following a NO_INLINE_CALL directive, the Application Compiler generates an error message.

Like INLINE_CALL, the NO_INLINE_CALL directive can be followed by a set of parentheses containing a list of procedures that are called in the next statement. Without the parentheses and procedure list, NO_INLINE_CALL prevents the Application Compiler from inlining any calls made by the next statement. In the following code, for example, the NO_INLINE_CALL directive prevents the Application Compiler from inlining either call.

Fortran code
`C$DIR NO_INLINE_CALL` `      A=ROS(GAMMA)-SOR(GAMMA)`
C code
`#pragma _CNX no_inline_call` `   a=ros(gamma)-sor(gamma);`

When you add a set of parentheses containing a list of procedure names separated by commas, NO_INLINE_CALL affects only calls to the procedures in the list. In the following example, the NO_INLINE_CALL directive prevents the Application Compiler from inlining the calls to SOR, in the first statement, and to FN and FN1 in the second statement. The Application Compiler may or may not inline the calls to ROS, depending on the results of its profitability analysis.

Fortran code

`C$DIR NO_INLINE_CALL(SOR)`
`      A=ROS(GMA)*ROS(DL)-SOR(GMA)`
`C$DIR NO_INLINE_CALL(FN, FN1)`
`      B=ROS(DL)/FN(THETA)-FN1(B)`

C code

`#pragma _CNX no_inline_call(sor)`
`   a=ros(gma)*ros(dl)-sor(gma);`
`#pragma _CNX no_inline_call(fn, fn1)`
`   b=ros(delta)/fn(theta)-fn1(b);`

An inlining directive placed before a statement takes priority over an inlining directive placed before the body of a procedure. For example, a NO_INLINE_CALL directive takes priority over an INLINE directive.

The following example uses a NO_INLINE_CALL directive on a call to ROS, and an INLINE directive on the procedure itself. The NO_INLINE_CALL directive takes priority and causes the compiler to ignore the INLINE directive for that one specific call. The second call to ROS is still inlined.

Fortran code	C code

<pre>   MAIN
 	
      REAL A(1000), B
      INTEGER J
C$DIR NO_INLINE_CALL
      B=ROS(A(1))
      DO J=1,100
        A(J)=ROS(A(J))
      ENDDO
      END
 	
      FUNCTION ROS(X)
C$DIR INLINE
C function body
C omitted for clarity
      END</pre>
	<pre>main{
	float a[1000], b;
	int j;
	#pragma _CNX no_inline_call
	b=ros(a[1]);
	for (j=0; j<100; ++j)
	  a[j]=ros(a[j]);
	}
	
	ros(x) {
	 #pragma _CNX inline
	 /* function body
	    omitted for clarity
	 */
	}</pre>

If you place two inlining directives, one of which has a specific list of procedures, before a statement, the directive with the procedure list takes priority over the other directive. In the following code fragment, the NO_INLINE_CALL directive prevents the Application Compiler from inlining the calls to F1 and F3, but the INLINE_CALL directive takes priority for the specified procedure, F2. The Application Compiler inlines the call to F2.

Fortran code

```
      DO I=B,E
C$DIR INLINE_CALL(F2)
C$DIR NO_INLINE_CALL
        D(I)=F1(B,E)+F2(I,E)+F3(B,I)
      ENDDO
```

C code

```
for (i=b; i<=e; ++i)
   #pragma _CNX inline_call(f2)
   #pragma _CNX no_inline_call
   d[i]=f1(b,e)+f2(i,e)+f3(b,i);
```

Similarly, a NO_INLINE_CALL directive with a list of specific procedures takes priority over an INLINE_CALL directive, which may have a list of procedures. In the next example, the NO_INLINE_CALL directive prevents the Application Compiler

from inlining the calls to F1 and F2. The call to F3 is inlined because of the INLINE_CALL directive.

Fortran code

```
      DO I=B,E
C$DIR INLINE_CALL
C$DIR NO_INLINE_CALL(F1, F2)
         D(I)=F1(B,E)+F2(I,E)+F3(B,I)
      ENDDO
```

C code

```
for (i=b; i<=e; ++i)
   #pragma _CNX inline_call
   #pragma _CNX no_inline_call(f1, f2)
   d[i]=f1(b,e)+f2(i,e)+f3(b,i);
```

Cloning directives

The Application Compiler decides whether to clone a procedure by examining the way certain arguments are used on each call. By default, the cloning algorithm looks at arguments that are passed a constant value. The Application Compiler creates a clone to propagate one of these constants if doing so can improve optimization.

Even when these conditions are met, cloning does not always take place. Often, a procedure is inlined instead, eliminating the need for a clone. Cloning directives let you change the set of arguments that the cloning algorithm considers important.You can increase or decrease the number of clones using the -inline or -clone option, as described in Chapter 3, "Using the Application Compiler". Adding directives to a program can be time consuming, so if your goal is simply to increase or decrease the number of procedures cloned, use an option instead.

CLONE

The CLONE directive tells the Application Compiler to consider additional arguments when evaluating a procedure call for cloning. The CLONE directive is placed just before the procedure body, the same as an INLINE directive, and is used alone or with a list of argument names.

Without a name list, CLONE tells the Application Compiler to clone the specified procedure if a constant can be propagated to any of the arguments. The following example shows the CLONE directive used without an argument-name list. When CLONE is used in this manner, the Application Compiler creates a clone (or

reuses a clone made for another call) whenever cloning makes constant propagation possible, even if inlining is also possible.

In the example below, DFUN represents a procedure that the Application Compiler can inline. MAIN passes a constant to DFUN on the first call. The corresponding argument (N) is used to set a stop value for the J loop. The Application Compiler could create a clone of DFUN and propagate the constant 100 into the clone, but by default it prefers to inline DFUN instead. Because the CLONE directive is used, however, the Application Compiler prefers to clone DFUN whenever it can propagate any constant; it creates a clone of DFUN and propagates the constant 100 on the first call.

Fortran code	C code
`      MAIN`	`main {`
`      INTEGER I`	` int i;float a[50];`
`      REAL A(50)`	` ...`
`      ...`	` a[0]=dfun(100)/2;`
`      A(1)=DFUN(100)/2`	` for (i=1;i<50;i++)`
`      DO I=2,50`	`   a[i]=dfun(100+i);`
`       A(I)=DFUN(100+I)`	`}`
`      ENDDO`	
`      END`	`float dfun(n)`
	`int n;`
`      FUNCTION DFUN(N)`	`{`
`C$DIR CLONE`	` #pragma _CNX clone`
`      INTEGER N`	` int j;`
	` for (j=0;j<n;j++)`
`      DO J=1,N`	` {`
`      ...`	`  ...`
`      ENDDO`	` }`
`      ...`	` ...`
`      END`	`}`

The second call to DFUN, inside the I loop, passes a variable rather than a constant. Because there are no constants to propagate, the Application Compiler does not clone this call, even with the CLONE directive, but performs inlining instead. A NO_INLINE directive would have exactly the same effect as CLONE on the first call but would prevent inlining of the second call as well.

The next example shows a slightly different use of the CLONE directive. Here, the NO_INLINE directive does not result in cloning; you must use the CLONE directive instead.

Fortran code	C code
`      MAIN`	`main{`
`      INTEGER I`	` int i;float a[50];`
`      REAL A(50)`	` ...`
`      ...`	` a[0]=dfun(1)/2;`
`      A(1)=DFUN(1)/2`	` for (i=1;i<50;i++)`
`      DO I=2,50`	`   a[i]=dfun(i);`
`        A(I)=DFUN(I)`	
`      ENDDO`	`}`
`      END`	
`      FUNCTION DFUN(N)`	`float dfun(n)`
`C$DIR CLONE`	` int n;`
`      INTEGER N`	`{`
	` #pragma _CNX clone`
`C     N not used in`	` /*n not used */`
`C     any loop`	` /*in any loop*/`
`      ...`	` ...`
`      END`	`}`

Once again, DFUN represents a procedure that the Application Compiler would ordinarily inline. MAIN passes a constant to DFUN on the first call, but the corresponding argument (N) is not used in any loop. By default, the Application Compiler does not consider N to be important, and therefore does not clone to absorb this constant. When the CLONE directive is used, however, the Application Compiler does consider N to be important. It therefore creates a clone of DFUN so that it can propagate the constant 1 into that function. Inlining is not done.

Just as in the previous example, the second call to DFUN passes a variable rather than a constant. Nothing can be propagated, so the Application Compiler inlines the procedure for that call instead.

Programmers often use the CLONE directive to tell the Application Compiler to clone a procedure to propagate an additional constant. For example, arguments are sometimes used as switches to select between large sections of code within a procedure. If the argument variable can be replaced by a propagated constant, the compiler can eliminate unnecessary tests and remove large chunks of dead code. By default, however, the Application Compiler only clones to propagate constants that affect vectorization on C Series machines, or loop blocking on SPP Series machines.

Consider the following example. The Application Compiler does not automatically clone the call to FOO. The argument A does not

affect vectorization or blocking of any loops in FOO, so the algorithm does not consider cloning to be advantageous. Looking at the code, you can see that propagating the constant would allow the Application Compiler to eliminate large blocks of dead code. The CLONE directive tells the Application Compiler to create a clone if any constant can be propagated. This propagation enables the Application Compiler to eliminate three unnecessary IF tests and two large blocks of unreachable code.

Original code	Optimized code

<pre>   PROGRAM MAIN
      REAL AR1(500),AR2(500)
      CALL FOO(AR1,AR2,1)
      CALL FOO(AR1,AR2,2)
      END
 	
      SUBROUTINE FOO(AR1,AR2,A)
C$DIR CLONE
      REAL AR1(*),AR2(*)
      INTEGER A
      IF (A.EQ.1) THEN
        ! CODE BLOCK ONE
      ENDIF
      IF (A.EQ.2) THEN
        ! CODE BLOCK TWO
      ENDIF
      IF (A.EQ.3) THEN
        ! CODE BLOCK THREE
      ENDIF
      ...
      END</pre>
	<pre>PROGRAM MAIN
	REAL AR1(500),AR2(500)
	CALL FOO$CLONE$1(AR1,AR2,1)
	CALL FOO$CLONE$2(AR1,AR2,2)
	END
	
	SUBROUTINE FOO$CLONE$1(AR1,AR2,A)
	REAL AR1(*),AR2(*)
	INTEGER A
	
	! IF test eliminated
	! CODE BLOCK ONE
	
	! IF test eliminated
	! Code block eliminated
	
	! IF test eliminated
	! Code block eliminated
	END
	
	SUBROUTINE FOO$CLONE$2(AR1,AR2,A)
	REAL AR1(*),AR2(*)
	INTEGER A
	! IF test eliminated
	! Code block eliminated
	
	! IF test eliminated
	! CODE BLOCK TWO
	
	! IF test eliminated
	! Code block eliminated
	END</pre>

The CLONE directive can be followed by a set of parentheses enclosing a list of argument names. Used with such a list, the CLONE directive tells the Application Compiler to clone the procedure if any of the listed arguments are constants. Other

arguments are evaluated normally; the Application Compiler creates a clone to absorb them only if they affect vectorization or blocking of loops.

For example, the CLONE directive in the following code tells the Application Compiler to make a clone of DFUN if doing so enables it to propagate the character variables M1 or M2 (as in the first call to DFUN from MAIN). The directive does not affect the integer N, however. If M1 and M2 cannot be propagated, as in the second call, the Application Compiler clones DFUN only if N affects the vectorization or blocking of a loop and DFUN cannot be profitably inlined.

Fortran code

```
      PROGRAM MAIN
      CHARACTER CVAR,CVAR1
      ...
      CALL DFUN("A",CVAR1,50)
      CALL DFUN(CVAR,CVAR1,50)
      ...
      END

      SUBROUTINE DFUN(M1,M2,N)
C$DIR CLONE(M1,M2)
      CHARACTER M1,M2
      INTEGER N
      ...
      END
```

C code

```
main(){
 char cvar,cvar1;
 ...
 dfun('A',cvar1,50);
 dfun(cvar,cvar1,50);
 ...
}

void dfun(m1,m2,n)
char m1, m2;
int n;
{
 #pragma _CNX clone(m1,m2)
 ...
}
```

Although the CLONE and INLINE directives specify two seemingly opposite optimizations, you can use them together.

The CLONE directive overrides the INLINE directive whenever propagation is possible, and the Application Compiler generates a warning telling you that this has happened. If a call does not pass a constant to any of the variables affected by the CLONE directive, the INLINE directive takes effect and forces the Application Compiler to inline the call.

Consider the following example. This function has two character arguments, M1 and M2, and one integer argument, N. By default, the Application Compiler clones this procedure only if it sees that N is used in a loop. Because of the CLONE directive, however, the Application Compiler creates a clone whenever it sees a chance to propagate a constant to M1 or M2. If neither M1 nor M2 is a constant on some particular call, the INLINE directive overrides any consideration of the integer N, and the procedure is inlined instead.

Fortran code	**C code**

`      FUNCTION DFUN(M1,M2,N)`
`C$DIR CLONE(M1,M2)`
 	
`C$DIR INLINE`
`      INTEGER M1,M2,N`
`       ...`
	`float dfun(m1,m2,n)`
	`int m1,m2,n;`
	
	`{`
	`#pragma _CNX clone(m1,m2)`
	`#pragma _CNX inline`
	`...`

NO_CLONE

The NO_CLONE directive tells the Application Compiler not to consider constants passed to certain arguments as a justification for cloning a procedure. If the cloning of a particular procedure causes a significant increase in your program's size, and you do not believe the increased speed due to cloning justifies the additional memory usage, consider using this directive. This is especially true if you notice an increase in memory swapping when you execute your program after compiling with the Application Compiler.

Like the CLONE directive, NO_CLONE is placed just before the procedure body and can be used alone or with a list of argument names. The following example shows the NO_CLONE directive in its simplest form. When no arguments are specified, NO_CLONE prevents the Application Compiler from cloning because of any argument.

Fortran code	C code

`      FUNCTION DFUN(A,B,C,N)`
`C$DIR NO_CLONE`
`      REAL A(*),B(*),C(*)`
`      INTEGER N`
`      ...`
	`float dfun(a,b,c,n)`
	`float a[],b[],c[];`
	`{`
	`#pragma _CNX no_clone`
	`integer n;`
	`...`

If you list argument names, the NO_CLONE directive prevents the Application Compiler from creating a clone to propagate any of the listed arguments. The Application Compiler can still clone the procedure to propagate a constant to an argument you haven't applied the NO_CLONE directive to. In that case, it can also propagate a constant passed to an argument that does have the NO_CLONE directive applied. For example, in the following code fragment, the NO_CLONE directive tells the Application Compiler not to clone the procedure in order to propagate a constant passed to L, even if L is used within a loop. The Application Compiler may still clone the procedure to propagate M or N, however, and, if this is done, it may propagate L also.

Fortran code	C code

`      FUNCTION DFUN(L,M,N)`
`C$DIR NO_CLONE(L)`
`      INTEGER L,M,N`
`      ...`
	`float dfun(l,m,n)`
	`{`
	` #pragma _CNX no_clone(l)`
	` integer l,m,n;`
	` ...`

You can use a NO_CLONE directive together with a NO_INLINE directive to prevent a procedure from being cloned or inlined. The NO_CLONE directive does not increase the chances of inlining since the Application Compiler prefers inlining by default.

Other directives

These directives give the Application Compiler additional information, allowing it to make better choices for optimizing or error checking, or instructions on how to link the code.

- CONDITION_TRUE
- ESTIMATED_TRIPS
- FORCE_OBJECT
- VAR_ARGS

CONDITION_TRUE

The Application Compiler makes decisions about inlining code based on two criteria: the size of the procedure being called and the frequency of the call. Other optimizations also depend on the frequency of specific calls. Determining these frequencies is not a trivial process. To properly determine these frequencies, the analysis algorithm must take into account every branching and every conditional statement that affects the flow of control to a procedure call. For each conditional statement, the algorithm must estimate the probability of execution for each branch. Sometimes the condition being tested is a constant that can be evaluated at compile time. When this happens, the outcome is known and the conditional can be optimized away. More often, the Application Compiler must assume that both possible branches from a conditional are equally likely.

You can improve the efficiency of the Application Compiler's analysis by providing information about the probability of a conditional's outcome. To do this, use the CONDITION_TRUE directive. The CONDITION_TRUE directive is placed on the source line immediately before the conditional statement. It is followed by a set of parentheses that contains an integer from 1 to 99. This number is your estimate of the probability that the outcome of the condition will be true.

The following code fragment shows the use of the CONDITION_TRUE directive. The programmer suspects that the value of DELTA will be greater than PI for 99% of all cases. The CONDITION_TRUE statement tells the Application Compiler the call to ROTATE will be made 99 out of every 100 times this statement is executed. This information allows the Application Compiler to make a better decision about the profitability of inlining the call.

Fortran code	C code

`C$DIR CONDITION_TRUE(99)`
`      IF (DL.GT.PI) THEN`
`         CALL ROTA(FL,DL)`
`      ENDIF`
	`#pragma _CNX condition_true(99)`
	`if (delta>pi)`
	`   rota(fl,dl);`

In the next example, the CONDITION_TRUE directive tells the Application Compiler that A is equal to B(1) only 10% of the time. This allows the Application Compiler to make a better decision about the profitability of inlining MEASURE.

C code

```
#pragma _CNX condition_true(10)
if (a == b[1])
  {
    a=0;
    measure (a,b);
  }
```

Fortran code

```
C$DIR CONDITION_TRUE(10)
      IF (A.EQ.B(1)) THEN
        A=0
        CALL MEASURE(A,B)
      ENDIF
```

The CONDITION_TRUE directive cannot be used on a WHILE or while statement, on a Fortran three-way IF statement that has three distinct target labels, or on a C switch statement.

ESTIMATED_TRIPS

ESTIMATED_TRIPS tells the Application Compiler how many times you expect a loop to iterate. The Application Compiler uses this information to make better decisions about inlining calls made within the loop. The ESTIMATED_TRIPS directive serves much the same role as the CONDITION_TRUE directive.

The following code fragment shows the use of the ESTIMATED_TRIPS directive. Suppose you think that the I loop will typically make about 200 iterations each time it executes. The Application Compiler cannot determine the trip count for this loop because the values of N and M are unknown at compile time. By using an ESTIMATED_TRIPS directive as shown here, you can tell the Application Compiler how often you expect the loop to

iterate and, therefore, how often the call to SBFN will be made. The Application Compiler decides whether to inline the call based on this information and the length of procedure SBFN.

C code

```
#pragma _CNX estimated_trips(200)
for (i=0; i<n; i+=m)
  sbfn(i,m);
```

Fortran code

```
C$DIR ESTIMATED_TRIPS(200)
      DO I=1,N,M
        CALL SBFN(I,M)
      ENDDO
```

Estimates need not be exact to be useful. If profiling data indicates a wide range of trip counts for a loop, choose the median value.

FORCE_OBJECT

The FORCE_OBJECT directive forces the Application Compiler to completely compile and link a procedure into the executable, even if interprocedural analysis shows that the procedure is never called. This directive performs the same function as the FORCE_OBJECT buildfile command described in Chapter 4. Like the buildfile command, it is useful when you have functions, such as conversion routines, that are not called directly by your program but are passed to library routines that use them. Unlike the buildfile command, the FORCE_OBJECT directive takes no arguments; the procedure to which it applies can be automatically determined by the directive's location.

The FORCE_OBJECT directive is used like this:

C code

```
char foo(char a) {
#pragma _CNX force_object
...
```

Fortran code

```
      FUNCTION FOO(A)
C$DIR FORCE_OBJECT
...
```

VAR_ARGS

The VAR_ARGS directive in a source file performs the same function as the PSUM_VAR_ARGS directive in a PSUM file or a var_args statement in a buildfile: given a procedure that can take a variable number of arguments or parameters, it tells the Application Compiler how many arguments or parameters to consider significant for error checking. If the Application Compiler finds that a call does not pass enough arguments or parameters to a procedure, it issues a warning. Unless you tell it otherwise, the Application Compiler assumes all arguments or parameters listed in a procedure declaration are required.

The following example shows Fortran and C procedures having five arguments or parameters. Only the first two are mandatory, however. The VAR_ARGS directive tells the Application Compiler not to consider a call that leaves off one of the optional arguments or parameters to be an error

.

C code

```
void sub1(a,b,c,d,e);
char *a,*b,*c,*d,*e;
{
/* indicate only 2 args req'd for sub1: */
 #pragma _CNX var_args(2)
  .
  .
  .
}
```

Fortran code

```
      SUBROUTINE SUB1(A,B,C,D,E)
      CHARACTER A,B,C,D,E
C$DIR VAR_ARGS(2)  ! INDICATES ONLY 2 ARGUMENTS
                   ! REQUIRED FOR SUB1
      .
      .
      .
      END
```

You can also use the VAR_ARGS directive without an argument to tell the Application Compiler that none of the arguments are mandatory. In the following example, the VAR_ARGS directive tells the Application Compiler to consider all of the arguments to SUB1 optional:

Fortran code

```
      SUBROUTINE SUB1(A,B,C,D,E)
      CHARACTER A,B,C,D,E
C$DIR VAR_ARGS
      .
      .
      .
      END
```

C code

```
void sub1(a,b,c,d,e);
char *a,*b,*c,*d,*e;
{
 #pragma _CNX var_args
  .
  .
  .
}
```

Optimization strategy

7

The *Convex Fortran Optimization Guide* and *Convex C Optimization Guide* present strategies for optimizing your code to run on a Convex C Series supercomputer. With the Application Compiler, the strategy is modified to take advantage of interprocedural optimization. This chapter presents a strategy for optimizing C and Fortran programs using the Convex Application Compiler running on either C Series or SPP Series machines.

The Application Compiler automatically analyzes all the procedures in your program, making many of the directives required by the procedural compilers unnecessary. This makes program optimization simpler, safer, and less time consuming.

For loop-intensive programs that manipulate large arrays and matrices, the greatest performance gains usually come from vectorization on C Series machines, and from data-localization and large-scale parallelization on SPP Series machines. To achieve the best performance, you must vectorize or parallelize the loops that take the most execution time. C Series machines can sometimes achieve even greater performance by parallelizing vectorized loops. By resolving apparent recurrences that prevent vectorization and parallelization, the Application Compiler ensures that the maximum number of loops are vectorized and parallelized.

Note

When you are optimizing code, it is 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. Test your code at each stage of the optimization process to make sure that the optimized program still gives correct results.

Step 1: Scalar optimization

If you are compiling a new or modified program, the first step is to produce a scalar executable for debugging. If you are recompiling an existing vectorized Convex C or Fortran program on a C Series machine to gain additional performance, proceed directly to Step 3.

Step 1a: compile at `-O1`

Compile the program with scalar optimization (compiler optimization level `-O1`). The additional error checking the Application Compiler does makes it much less likely that a bug will slip through and makes it easier to isolate problems. Use `-check all` on the `apc` command line.

If you have access to a profiler such as `prof`, `bprof`, `gprof`, or CXpa, use the appropriate compiler option on the compilation line in your buildfile to facilitate profiling for each procedure. The compiler options which allow profiling with each of these products are described in the *Fortran User's Guide* and the *C User's Guide.*

Note

Automatic inlining, as performed by the Application Compiler, sometimes causes programs to fail when run under CXpa. If you experience this problem, recompile your code with `-inline none` or use another profiler.

If you have access to a debugger such as CXdb (available on both SPP and C Series machines) or `csd` (C Series only), you may also consider including the appropriate compiler option to facilitate debugging.The compiler options which allow debugging with each of these products are described in the *Fortran User's Guide* and the *C User's Guide.*

Note

If you use the `csd` debugger, you may notice warnings about multiple source modules with the same name. These warnings occur because the Application Compiler generates multiple object files for each source file. The warnings are harmless and can be ignored.

When compilation is complete, note any error messages the Application Compiler generates. Find the source code errors responsible for these messages and repair the code. Note any warning messages the Application Compiler generates. Either fix the code responsible for these warning messages or save the messages to refer to in case you run into problems later on. You can easily save such messages from within the PDB Viewer; refer to Appendix A, "The PDB Viewer interface," for details.

Step 1b: check output

Run the executable you have produced and check the output against expected values. If you are porting the program from another machine, compare the output of the new executable with the output from a test run of the program on the other machine.

If the output matches the expected results (be sure to allow for roundoff differences), go to Step 2. If it does not, debug your program before proceeding to the next step. Recompile the program using `-show renames`, `-show all`, `-check all`, and the your chosen debugger option. Study any warning messages the Application Compiler generates as clues to the location of possible problems. Look for obvious errors in your source code. If you do not find any errors, try the following steps.

1. Reduce the level of inlining (`-inline low`, `-inline none`) and recompile.
2. Reduce the level of cloning (`-clone none`) and recompile.
3. Reduce the optimization level (to `-O0` or `-no`) and recompile.
4. Disable interprocedural optimizations with the `-no ipo` option and recompile.

After each step, check your output to see if the problems persist. If you cannot isolate the problem this way, recompile your program at `-no` using either the Application Compiler or the conventional compiler. Use a debugger flag to include debugging information, then use your debugger to pinpoint and fix logic errors that may be causing the problem. If you cannot find a logic error, contact the Convex Technical Assistance Center as described in "How to use this book" to report a possible compiler bug.

Note

If your program produces different answers when compiled with the Application Compiler, do not automatically assume that the new answers are wrong. Interprocedural optimization sometimes allows programs to run with a higher degree of precision, producing better answers than those achieved by code compiled with conventional compilers. Interprocedural optimization can also reveal numerical instabilities in an algorithm, which must be corrected to ensure correct performance under all conditions. Complete understanding of your algorithm can help you determine whether the answers are correct.

Scalar optimization rarely affects the output of a program, so it is unlikely that you will find significant differences between `-O1`, `-O0`, and `-no`. If you do, use a binary search to isolate the procedure responsible for the change in your program's output.

Compile half the procedures at `-no` and the other half at `-O1`. Run the program and check the output to determine which half contains the procedure causing the problem. If the output is now correct, the bad procedure is in the half compiled at `-no`. Next, split the suspect group in half. Again, compile half of the suspect procedures at `-no` and the other half at `-O1`. Continue bisecting the suspect group in this manner until you isolate the offending procedure.

Unless you are compiling code that has been previously optimized above the `-O1` level using the Convex C or Fortran compiler, do not skip this first step. Optimizations performed at higher levels make debugging much more difficult. Be sure your program produces correct results before you start to add optimization.

Step 1c: profile your program

Compile your program with the appropriate procedural compiler. Run the resulting executable and obtain a profile using your profiler of choice. Note the program's total execution time and the procedures that use the most time.

Step 2: `-O2` optimization

The `-O2` optimization level provides data localization optimizations. On C Series machines, data localization involves vectorizing loops wherever possible. On SPP Series machines, it involves making optimal use of the processor data cache. For more information on data localization, refer to the *Exemplar Programming Guide*; for more information on vectorization, refer to the *Fortran Optimization Guide* or the *C Optimization Guide*.

The Convex C Series optimization guides recommend two possible approaches to `-O2` level optimization; one involves compiling the entire program at `-O2`, the other involves compiling only the most time-consuming procedures at `-O2` first, and proceeding with optimization using a step-by-step approach. When you use the Application Compiler, interprocedural analysis automatically resolves many common optimization problems, so this measured, step-by-step approach to `-O2` optimization is unnecessary regardless of target hardware.

Step 2a: compile at `-O2`

Using the Application Compiler, compile the entire program at optimization level `-O2`. Use a profiler option as described in Step 1.

Do not try to apply level -O2 optimizations to a procedure that produces incorrect results at optimization level -O1. The compiler continues to perform scalar optimizations at -O2, so any problems that you encounter at -O1 will recur when you proceed to level -O2.

Step 2b: check output

Run the -O2 version of your program and compare the output with the output produced in Step 1. If the two outputs match (be sure to allow for roundoff differences), continue on to Step 2c. If the outputs do not match, debug your program before proceeding to the next step. Again, study any warning messages that the Application Compiler generated in Step 1 as clues to the location of possible problems.

Use the binary search procedure described in Step 1 to isolate the procedure responsible for the change in your program's output. When you have done so, check its source code and fix any errors that you find. If you do not find any logic errors, recompile that subprogram at -O1 and continue optimizing the rest of the program.

Step 2c: profile your program

Study the execution profile. Compare this profile with the profile obtained in step 1c. Usually, you will find that all procedures in your program speed up when the program is compiled at -O2. If a particular procedure contains code that is inherently scalar, performance may improve slightly or not at all. If a procedure slows down when level -O2 optimizations are applied, carefully examine its algorithm and attempt to determine the cause of the slowdown. If the cause cannot be found, try rewriting the procedure using a different approach, or, if its level -O1 performance is acceptable, forego -O2 optimizations for that particular procedure.

Caution

It is always possible that rewriting code may not improve performance; it may even lower it. Always save a backup copy before rewriting code.

Step 2d: look for further improvements

Examine the optimization report and the loop-level profiles to determine which loops were successfully data-localized and

which transformations were performed. Carefully examine the procedures with the longest execution times, looking for opportunities to improve data localization. On C Series machines, conditional statements sometimes prevent vectorization of a loop. If a loop containing a conditional statement does not vectorize, try rewriting the code to remove the conditional.

Candidate loops for data-localization on SPP Series machines must exhibit a number of characteristics. If a procedure is failing to make optimal use of the processor data cache on a SPP Series machine, refer to the *Exemplar Programming Guide* to determine if the procedure can be modified to better take advantage of data localization.

Even if the Application Compiler has data-localized all possible loops, you may be able to further improve the performance of your code. Try the following optimizations:

- On C Series machines, try compiling your code using the `-ds` (dynamic selection) compiler option (this is the default on SPP Series machines). Some programs compiled with the Application Compiler show significant performance gains when dynamic selection is enabled. Some code may slow down due to dynamic selection, however. Be sure to get profiles and compare timings to determine the best version of your code.
- Simplify conditionals. An embedded conditional always slows a loop's performance even if the loop is data-localized. If the embedded conditional can be simplified or removed, you can enhance the loop's performance.
- Simplify array subscripts. Complicated subscripts always slow down the execution of a loop, even at level `-O2`. Make the subscripts into your arrays as simple as possible.
- In code running on C Series machines, watch for loops with short vector lengths (small iteration counts). Loops with small iteration counts usually run faster in scalar form. If a loop's iteration count is always less than five and the loop is vectorized, use the `SCALAR` directive on the loop to prevent vectorization.
- If you profiled your program as discussed in Step 2c, use the appropriate option to `apc` (`-prof` or `-gprof` on C Series or `-pdf` on SPP Series machines) to feed the generated profile data back to the APC to further improve optimization efficiency.
- If compiling your program with the Application Compiler has increased the size of your executable, watch for indications of increased paging. If excessive paging appears to be slowing your program down, you may want to try

decreasing the amount of inlining or cloning using the command line options (-inline low, -clone none), source directives (NO_INLINE, NO_INLINE_CALL, and NO_CLONE), or buildfile statements (no_inline and no_clone).

- Examine the flow of control in each procedure. Look for conditionals in which one branch is taken much more often than the other. Use a CONDITION_TRUE directive to tell the Application Compiler how often to expect the TRUE branch to be taken. This allows the Application Compiler to better analyze the procedure and to generate better optimizations.
- Look at the messages the Application Compiler generates and note which procedure calls are inlined and parallelized. Look for additional calls that you think might profitably be inlined or parallelized and for calls that are inlined or parallelized but perhaps should not be. The Application Compiler analyzes eligible procedure calls based on size, call frequency and data use to determine whether to inline them or parallelize their calls.While the Application Compiler can always determine the procedure size, it cannot always determine how frequently a call is made. This is true when the frequency of a call is determined by an input value or a conditional whose value cannot be determined at compile time.

  If you can estimate the frequency of such a call and the estimated frequency is low, you can use a NO_INLINE_CALL directive to prevent inlining. If the estimated frequency is high, you can force inlining using an INLINE_CALL directive. (Do not force inlining of a procedure containing hundreds of lines of code no matter how frequently it is called; such procedures should have their calls parallelized if at all possible.) If a conditional statement determines the frequency of the procedure call, you can use a CONDITION_TRUE directive instead to let the Application Compiler make its own decision based on your information.

Step 2e: recompile

Recompile your program and profile it again. Compare the output with the previous output and make sure it has not changed. Examine the execution profile and compare the timings for the procedures you worked on. If a procedure slowed down because of the changes you made, remove the changes. If a procedure still runs slower at -O2 than at -O1, recompile the procedure at -O1.

Step 3: -O3 optimization

The -O3 optimization level parallelizes your code where possible. Unlike data localization, parallelization does not reduce a program's CPU time. By spreading a program's work across multiple CPUs, parallelization can reduce the time a program takes to arrive at a solution. In the best case, parallelization can improve a program's time to solution by a factor of N, where N is the number of available CPUs on your machine. Because of overhead, total CPU time typically increases slightly when a program is parallelized. Programs running on SPP Series machines realize the greatest gains from parallelization because they typically have many more processors than C Series machines.

Not all programs can achieve the theoretical factor-of-N speedup. Many programs have algorithms that cannot be parallelized or slow down when parallelized. Other programs may not benefit from parallelization because they run on heavily loaded machines where the system load prevents an application from receiving more than one CPU at any given time; this is a bigger consideration on C Series machines. If you believe that your program could benefit from parallelization given the hardware and number of CPUs available to your program, follow the procedures in this step to parallelize your program. Otherwise go on to Step 4.

Step 3a: compile at -O3

Using the Application Compiler, compile the entire program for parallelization (optimization level -O3). Use the -pdf or profiler option as described in Step 2.

Do not try to parallelize a procedure that produces incorrect results at lower optimization levels. The compiler continues to perform -O1 and -O2 optimizations at -O3, so any problems that remain unresolved at lower optimization levels will recur when you add parallelization. Compile those problem procedures at the highest optimization level that produced correct results.

Step 3b: check output

Run the parallelized version of your program and compare the output with the output produced in Step 2. If the two outputs match (be sure to allow for roundoff differences), continue on to step 3c. If the outputs do not match, debug your program before proceeding. Once again, study any warning messages the Application Compiler generates as clues to the location of possible problems. Use the binary search procedure described in Step 1 to

isolate the procedure responsible for the change in your program's output. When you have isolated the procedure causing the problem, check its source code and fix any visible errors. If you do not find any logic errors, recompile the offending procedure at -O2 and continue optimizing the rest of the program.

Step 3c: profile your program

Study the execution profile. Compare this profile with the profile obtained in Step 2. Note the process virtual time for each procedure. You may find that some procedures have slowed down due to parallelization. Recompile those procedures at -O2.

Step 4: Wrapping up

The -pdf and -p options cause the compiler to insert timing code and data, also called instrumentation, into your program. When the program is fully optimized, recompile without these options to remove the instrumentation overhead. Use the apc -c option to clean up the program database.

Fortran hints

8

This chapter presents some programming tips that are of particular use to Fortran programmers using the Application Compiler. The material here can help you obtain the best performance from your Fortran code and minimize unexpected problems.

Initialize COMMON-block variables

Some programs assume that Fortran automatically initializes variables in COMMON blocks to zero. The ANSI Fortran standard does not require automatic initialization of COMMON variables, although Convex Fortran does provide it. Any programs that depend on automatic initialization of COMMON variables are therefore nonstandard and nonportable. If you compile such a nonportable program with the `-check init` option, the Application Compiler reports the use of a COMMON block variable that is not explicitly initialized as a nonfatal error.

For example, the following Fortran code relies on the automatic initialization of arrays `A` and `IA`:

```
PROGRAM MAIN
REAL B(1000)
REAL A(1000)
INTEGER IA(1000)
COMMON /CM/ A, IA

CALL SUB1(B)

END
```

```
SUBROUTINE SUB1(B)
REAL B(*)

REAL A(1000)
INTEGER IA(1000)
COMMON /CM/ A, IA

DO I=1,1000
   B(I)=A(I)*IA(I)
ENDDO

END
```

The Application Compiler generates the error messages shown in Figure 46.

```
MAIN uses A, which is neither assigned nor initialized
MAIN uses IA, which is neither assigned nor initialized
SUB1 uses A, which is neither assigned nor initialized
SUB1 uses IA, which is neither assigned nor initialized

 ...

Errors Detected
---------------
                       Wrong  Mis-            Scalar
             Mis-      Number Matched         Passed           Variables
             Matched   Of     Return  Invalid To     Invalid   Not
Procedure    Args      Args   Type    Aliases Array  Subscript Initialized
------------ -------   ------ ------- ------- ------ --------- -----------
main                                                                     2
sub1                                                                     2
Totals                                                                   4
------------ -------   ------ ------- ------- ------ --------- -----------
build -check type      type   type                   arrays    init
```

Figure 46 Array initialization error messages

To ensure portable code, fix these errors by creating a BLOCK DATA subprogram to initialize all your COMMON-block variables (or use an assignment in executable code):

```
PROGRAM MAIN
REAL B(1000), A(1000)
INTEGER IA(1000)
COMMON /CM/ A, IA

CALL SUB1(B)

END


BLOCK DATA SUB1BLOCK
REAL A(1000)
INTEGER IA(1000)
COMMON /CM/ A, IA
DATA A/1000*0.0/, IA/1000*0/
END


SUBROUTINE SUB1(B)
REAL B(*)

REAL A(1000)
INTEGER IA(1000)
COMMON /CM/ A, IA

DO I=1,1000
B(I)=A(I)*IA(I)
ENDDO

END
```

Watch for dynamic binding

Binding is the process in which attributes or properties such as type and size are attached (bound) to a variable. In compiled languages, binding usually occurs at compile time. This is called static binding, because the attributes are fixed or static and cannot be changed at runtime. When attributes are bound to a variable at runtime, this is called *dynamic binding*, meaning that the attributes can change according to the program's execution.

The Application Compiler uses dynamic binding to inline a procedure that has a dummy argument with a different set of dimensions than the actual argument. For example, an array declared as A(10,10,10) passed to a procedure where the dummy argument is declared as A(1000). The array is

dynamically bound to the new shape and size. Dynamic binding results in additional overhead that reduces the benefit of inlining. Inlined routines that use dynamic binding usually (not always) run slower than the original, noninlined routines. For this reason, the Application Compiler does not inline most routines that use dynamic binding unless you specifically request that it do so.

The following sample code shows a matrix multiply routine with dynamic binding of arguments.

```
      PROGRAM MAIN
      REAL A(NUM,N0),C(N1),B(N2)

      ...
      CALL MATMUL(A(1,I),B,C,ND,ND,1)
      END


      SUBROUTINE MATMUL(A,B,C,L,M,N)
      DIMENSION A(L,M), B(M,N), C(L,N)
      DO 40 K=1,N
         DO 10 I=1,L
            C(I,K)=0.0
10       CONTINUE
         DO 30 J=1,M
            TEMP=B(J,K)
            IF (TEMP .EQ. 0.0) GOTO 30
            DO 20 I=1,L
               C(I,K)=C(I,K)+A(I,J)*TEMP
20          CONTINUE
30       CONTINUE
40    CONTINUE
      RETURN
      END
```

`MAIN` declares `B` as a one-dimensional array of length `N2`, but the subroutine `MATMUL` rebinds it to a two- dimensional of size `M` times `N`. This dynamic binding prevents the Application Compiler from inlining the matrix multiply unless you use `-permit dynamic_binding`.

The `-permit dynamic_binding` option tells the Application Compiler to inline procedures that use dynamic binding if they pass the compiler's usual compatibility test. The burden then falls on you to make sure that the newly inlined procedures do not slow your program down.

One specific case of dynamic binding never slows an inlined procedure down. When an array with more than one dimension is rebound to an array with one dimension, such as in the following

example, inlining does not produce any additional overhead. If you have many procedure calls of this type, consider using `-permit dynamic_binding`.

```
PROGRAM MAIN
REAL R(40,40,100)
CALL INIT(R)
END

SUBROUTINE INIT(R)
REAL R(160000)

DO I=1,160000
R(I)=1.0
ENDDO
END
```

One common instance of dynamic binding occurs in the use of assumed-size arrays. The dynamic binding caused by the assumed size array prevents the Application Compiler from inlining the following routine unless you use `-permit dynamic_binding`.

```
SUBROUTINE SUB1 (B,M,N)
REAL B (60,*)
PRINT *,B(M,N)
RETURN
END
```

If you rewrite the procedure without the assumed-size array, as shown below, the need for dynamic binding is eliminated and the Application Compiler inlines it by default.

```
SUBROUTINE SUB1A (B,M,N)
REAL B (60,4)
PRINT *,B(M,N)
END
```

The Application Compiler does not inline the following procedure by default because of the dynamically bound adjustable array.

```
SUBROUTINE
SUB2(B,M,N)
REAL B(M,N)
PRINT *,B(M,N)
END
```

Adjustable and assumed-size arrays of one dimension do not cause dynamic binding to slow down code. The Application Compiler inlines these procedures by default:

```
SUBROUTINE SUB3 (B,M)
REAL B (M)
PRINT *,B(M)
END

SUBROUTINE SUB4 (B,M)
REAL B (*)
PRINT *,B(M)
END
```

Watch for misused Hollerith constants

The ANSI Fortran standard prohibits passing Hollerith constants to CHARACTER variables.

The conventional Convex Fortran compiler allows this nonstandard practice, under some circumstances, and some existing programs may make use of it. For example, the following code passes the Hollerith constant 6HSTRING to subroutine PRINT_STRING, which takes a CHARACTER parameter.

```
PROGRAM MAIN
CALL PRINT_STRING(6HSTRING)
END

SUBROUTINE PRINT_STRING(S)
CHARACTER*(*) S
PRINT *, S
END
```

As stated in the *Fortran User's Guide,* a dummy argument that receives a Hollerith actual must have a numeric type. The Application Compiler sees that the call to PRINT_STRING does not pass a length specifier, and issues a warning that the number of actual arguments is less than the number of dummy arguments.

To fix this problem, change the argument from a Hollerith constant to a CHARACTER constant.

Watch array bounds

Declaring arrays with length 1, a common programming practice under Fortran 66, can cause the Application Compiler to report subscripting problems when you compile a program with -check arrays. Often, subroutines declare dummy array arguments as REAL A(1) or B(N,1) rather than REAL A(*) or B(N,*). Because storage has been allocated for the entire array in the original declaration, the subroutine can use subscripts greater than 1 to access the entire length of the array.

Because the declarations are identical, the Application Compiler cannot distinguish between arrays that are declared with length 1

so that they can be used as assumed-size arrays and arrays that really do have a length of 1. It therefore assumes that an error exists every time an array such as A, declared as A(1), is accessed with a subscript greater than 1. If the -check arrays option is not used, these errors are not reported, and you can compile your program without any problem. If the -check arrays option is used, however, apparent errors of this type may prevent compilation. There are three ways to solve this problem:

- Compile your program without the -check arrays option.
- Change the declaration of dummy arguments such as REAL A(1) or B(N,1) to REAL A(*) or B(N,*).
- Compile your program with the -a1 option, which causes the Application Compiler to ignore oversubscripting of arrays declared with a dimension length of one. (This is an fc compiler option. It can be used in the buildfile or assigned to an FFLAGS macro, but cannot be on the apc command line.)

The first method may cause the Application Compiler to overlook actual errors; the second requires changes to your source code. The third method, using -a1, is usually the easiest and ensures the safest code.

Watch for undefined dummy variables

The ANSI Standard says that dummy arguments not associated with actual arguments are undefined. On calls to ENTRY FOO1 in the following code, P2, P3, and P4 are undefined.

```
SUBROUTINE FOO(P1,P2,P3,P4)
INTEGER P1,P2,P3,P4
ENTRY FOO1(P1)
P2=P2+1
P3=P3+2
P4=P4+3
END
```

Some Fortran compilers, including the conventional Convex Fortran compiler, maintain the values of dummy arguments between calls. Thus, two calls made in the following sequence return the same results:

```
FOO(1,2,3,4)
FOO1(1)
```

The Convex Application Compiler does not maintain dummy values between calls. If your code relies on this nonstandard, implementation-dependent behavior, incorrect answers may result when you compile it with the Application Compiler. Rewrite the nonstandard code so that it does not rely on dummy values being maintained between calls.

Don't use `loc()` pointer functions

The Convex Application Compiler does not support Cray-style pointers or the use of the `loc()` function to take the address of a variable. If your program depends on these features, do not compile it with the Application Compiler. Doing so can result in wrong answers.

C language hints

9

This chapter presents some programming tips for C programmers using the Application Compiler. The material here can help you obtain the best performance from your C code and minimize unexpected problems.

Take advantage of pointer tracking

The Application Compiler's pointer-tracking algorithm, which is enabled by specifying the `-enable pointer_track` option on the `apc` command line, provides a more complete analysis of aliasing than previous methods do. More complete analysis of aliasing allows the Application Compiler to vectorize more loops. The pointer tracking algorithm can provide dramatic gains in performance without any need to rewrite existing code. By keeping a few things in mind, however, you can write code that makes pointer tracking more efficient. This can reduce compile times and result in faster code.

To ensure complete pointer tracking, declare all functions, including library functions, that return a pointer type. The following example has a call to `malloc` that is not declared by the program. This use of `malloc` defeats pointer tracking. Because there is no declaration, the Application Compiler does not know that `malloc` returns a pointer. The pointer returned by `malloc` is not included in pointer tracking, and aliasing may be missed.

```
main(){
  ...
  foo(malloc(100));
  ...
}
```

C hints

To solve this problem, provide a declaration for `malloc` as shown below:

```
char *malloc();
main(){
 ...
 foo(malloc(100));
 ...
}
```

Some programming styles make pointer tracking particularly easy. One of these styles declares all arrays to be one-dimensional. You can then use macros to do address calculations, as shown in the following example.

```
#define SUB(i,j,c) (i)*(c)+(j)
*(x+SUB(i,j,m))/* using macro defined above */
```

A related style defines multidimensional objects as structures. Again, you can use macros to access values within the structure. Like the previous style, this makes for more efficient pointer tracking.

```
typedef struct {
  int rows, cols;
  double * data;
} DM;

#define REF(x,i,j) x.data[j+i*x.cols]
```

Compiler options

A

With a few exceptions, the Application Compiler supports all compiler options provided by the current Convex Fortran and C compilers (versions 9.1 and 6.1, respectively) at the time of its release. For complete descriptions of the compiler directives, see the *Fortran User's Guide* or *C User's Guide*. Version 2.1 of the Application Compiler does not support options introduced in compilers introduced after Convex C version 6.1 and Convex Fortran V9.1.

The Application Compiler does not support the options listed in Table 11 provided by the current Convex Fortran or Convex C compiler. Do not use these options in your buildfiles.

Table 11 Unsupported Fortran and C options

Option	Languages
`-compat rrf=stack`	C
`-il`	Fortran
`-is`	Fortran
`-extern distinct`	C
`-extern same`	C
`-LST`	Fortran
`-LSTI`	Fortran
`-sl`	Fortran
`-string read_only`	C
`-string read_write`	C
`-xr`	C
`-xrl`	Fortran

The Application Compiler supports the options listed in Table 12 as command-line options to `apc` instead of compiler options. Use these options on the `apc` command line or in a buildfile `options` statement. Do not use them in a list of compiler options. (If the

Application Compiler finds a -o option on the link line, it silently moves the -o to the options line where it belongs.)

Table 12 Compiler options to apc

Option	Languages
-B	Fortran, C
-o	Fortran, C

The Application Compiler supports the fc and cc options listed in Table 13, but these options stop compilation early so that no executable is produced.

Table 13 Options that produce no executable

Option	Languages
-E	Fortran, C
-k	C
-sc	Fortran, C

The fc and cc options listed in Table 14 work with the Application Compiler. However, there is a difference in the way the resulting files are organized. Instead of naming the .s and .o files after source files, as the conventional compilers do, the Application Compiler names them by procedure. Also, instead of creating the files in the working directory, the Application Compiler creates them in the PDB directory. When used with the Application Compiler, -S allows a program to link; when used with fc or cc, it does not.

Table 14 Code generation options

Option	Languages
-S	Fortran, C
-c	Fortran, C

The Application Compiler supports the -D option. However, when you use this option in a buildfile, you must not use a blackslash before the double quotes surrounding the string constant, as required on the cc command line or in a makefile. See Table 15.

Table 15 -D option

apc	fc or make
-DDEF_1="foo"	-DDEF_1=\"foo\"

The PDB Viewer interface

B

The Convex PDB Viewer provides a graphical interface that allows you to view the optimization and error information in a form that is convenient and easy to understand. The Viewer, which is part of the Application Compiler package, works with program databases (PDBs) created by the APC.

Graphical user interface

The PDB Viewer uses Xwindows as its graphical user interface. Xwindows is an optional software product that must be installed in order for the PDB Viewer to run.

A number of different window managers are available for Xwindows, but the examples in this book use the TWM window manager.

All window managers allow you to size, iconify and kill windows. Unless otherwise noted, you can perform any operation supported by your window manager on any of the windows discussed in this appendix.

This appendix assumes that you are familiar with the basics of using Xwindows and your chosen window manager. For more information, refer to *The X Primer.*

Invoking the PDB Viewer

To invoke the Viewer, enter the command `pdbview`. This executable is typically found in the same directory as the `apc` executable (`/usr/convex/bin`). You can follow the `pdbview` command with any of the following options, which can also be configured as X Window resources.

`-B` *directory* or `-binDir` *directory*
: Use the alternate viewer executable files specified in the directory *directory.* Defaults to the directory in which the `apc` executable is installed, typically `/usr/convex/bin`. *directory* does not apply to `pdbview`, the viewer driver.

`-checkFilesTimes` or `-cf`
: Check the source files to see if they have been modified since the previous build.

`-contextRows` *rows* or `-cr` *rows*
: Set the number of lines around the call site in the Call Information window to *rows*. Defaults to 10.

`-graphLevels` *levels* or `-gl` *levels*
: Set the number of levels displayed in the call graph to *levels*. By default, all levels will be displayed.

`-help`, `-h`, or `-u`
: Display an online version of this list.

`-helpPath` *helpdir*
: Use the alternate viewer help files located in the directory *helpdir*. Defaults to `.../pdbviewlib/onlineHelp`, where `...` is the directory in which pdbview is installed.

`-path` *path*, `-pdb` *path*, or `-p` *path*
: Open the PDB contained in the directory specified in *path*.

`-prefetch` *type* or `-pre` *type*
: Open the initial main window specified in *type*. *type* can take the value `ipo_summary` (the default), `build_info`, or `outline`.

`-showXWarnings` or `-sxw`
: Allows the X server to write warning messages to `stderr`. The default is to disable X server messages.

`-sourceColumns` *cols* or `-sc` *cols*
: Set the number of columns of source displayed in the source pane of the Procedure Description window to *cols*. Defaults to 80.

`-sourceRows` *rows* or `-sr` *rows*
: Set the number of lines of source displayed in the source pane of the Procedure Description window to *rows*. Defaults to 20.

PDB Viewer

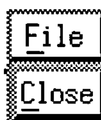
Common menus

Several menus appear in more than one of the PDB Viewer windows described in the following sections. These menus generally have common, if not identical, functions. Each menu that appears in a specific window is described in detail in that window's section when necessary. Descriptions of the common menus described in this section are omitted from the specific window sections that follow where possible.

The File menu

The File menu appears in every window generated by the PDB Viewer. All File menus (except the Main window's) contain at least a Close item which, when selected, closes the window in question. For many windows, Close is the only item in the File menu; elaborations on the File menu are omitted from sections covering these windows.

A minimal File menu is shown below. File menus containing entries in addition to Close are described in the appropriate following sections.



The Scratchpad menu

The Scratchpad menu appears in those PDB Viewer windows that contain information which may be useful to copy to an editable file. In windows that contain multiple panes of information, the Scratchpad menu generally contains an item that corresponds to each pane. Selecting one will copy the information contained in its corresponding pane to the scratchpad.

The scratchpad is a simple, window-based editor that accumulates the information copied to it and allows you to manipulate it and save it to a file if you wish. A Scratch Pad window containing a procedure outline is shown below.

```
File  View                                                            Help
                              Scratch Pad
Begin
    loop J = 1 to N step 1
        call MX1(&N-J+1, &A[JJ], &J-1, &LDA, &A[J1], &A[1J], &KK) [freq=15.0]
        loop I = ? to ? step ?
        end loop
        loop I = ? to ? step ?
        end loop
        call [inline] XM1(&N-J, &LDA, &A[J+1J], &J-1, &LDA, &A[1J], &LDA, &A[J+1
        loop I = ? to ? step ?
        end loop
    end loop
End
```

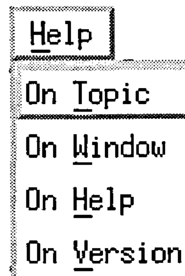
You can position the cursor by single clicking with the mouse or by using the arrow keys. The delete and backspace keys allow you to delete text, and can be used to delete blocks selected with the mouse.

The File menu allows you to Close the scratchpad window, in which case all information it contains is lost, or Save its contents to a file, which you will be prompted to name.

Selecting Clear from the View menu deletes all text from the scratchpad.

Help menu

The Help menu appears, with the same entries, in all PDB Viewer windows. A Help menu is shown below.

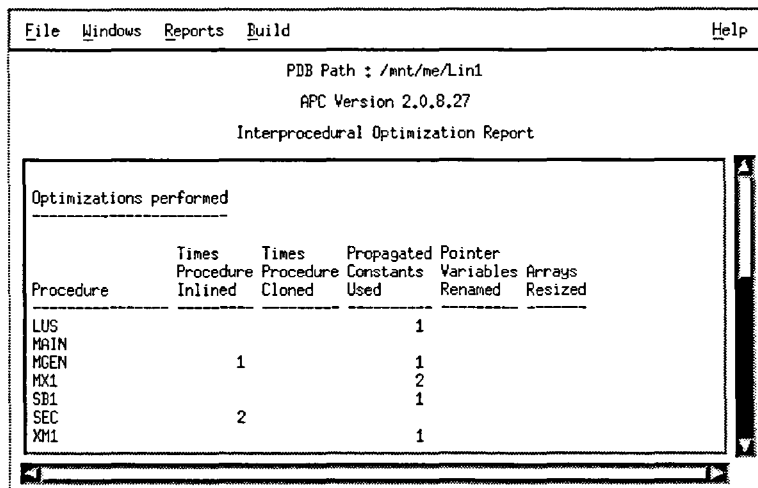


Selecting On Topic will display a window that allows you to choose a help topic. Selecting On Window will display the help window explaining the PDB Viewer window that is currently selected. Selecting On Help will display a help window to guide you through using the online help system. Selecting On Version

will display a window containing the current version of the PDB Viewer.

The main window

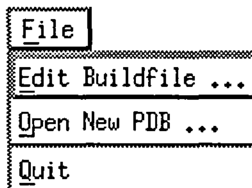
When you start the PDB Viewer, you see a main window like the one shown below (unless you have chosen a different main window using `-prefetch` option).



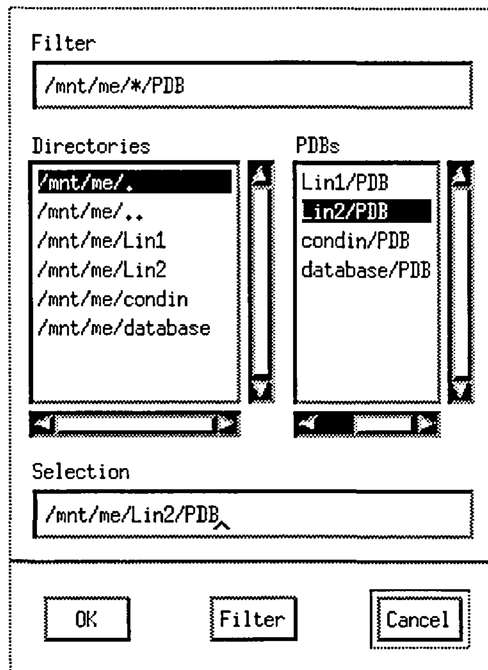
If you specified a PDB using the `-p` option, or if there is a PDB in the directory in which the PDB Viewer was started, the window displays the Interprocedural Optimization or IPO Report for that PDB, as shown here.

The File menu

If the main window appears blank, you must open a PDB by choosing Open New PDB from the File menu, which is shown below.



Choosing Open New PDB causes the file-selection dialog shown below to appear.



This dialog contains two lists: a list of directories and a list of PDBs found within those directories. To choose a PDB from the PDB list, simply click on the PDB you want, then click the `OK` button or press return. You can also choose a PDB by typing a pathname into the text box labeled Selection and clicking `OK` or pressing return.

The text box labelled Filter determines the PDBs displayed in the PDB list. To change the list, edit the contents of the Filter text box, using standard pathname syntax, to match the PDBs you want to see. For example, the text shown in the Filter box above matches any PDB in any subdirectory of `/mnt/me`. Then click on the Filter button; this action applies the filter, updating the PDB and Directories lists.

You can also navigate directories using the Directories list. Clicking on any directory within the Directories list puts that directory directly into the Filter text box, with `/*/PDB` appended to it. For example, clicking on /mnt/me/condin puts `/mnt/me/condin/*/PDB` into the Filter text box. Clicking on `/mnt/me/..` puts `/mnt/*/PDB` into the box. You can then move to the directory in the box by clicking on the Filter button.

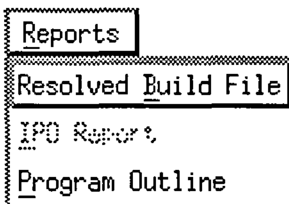
You can edit the current buildfile by choosing Edit Buildfile from the File menu. This will run the editor specified in your `$EDITOR`

environment variable in a new window, and load the buildfile into the editor.

Lastly, the File menu allows you to quit the PDB Viewer any time there isn't an open dialog. Choose the Quit item to quit the viewer.

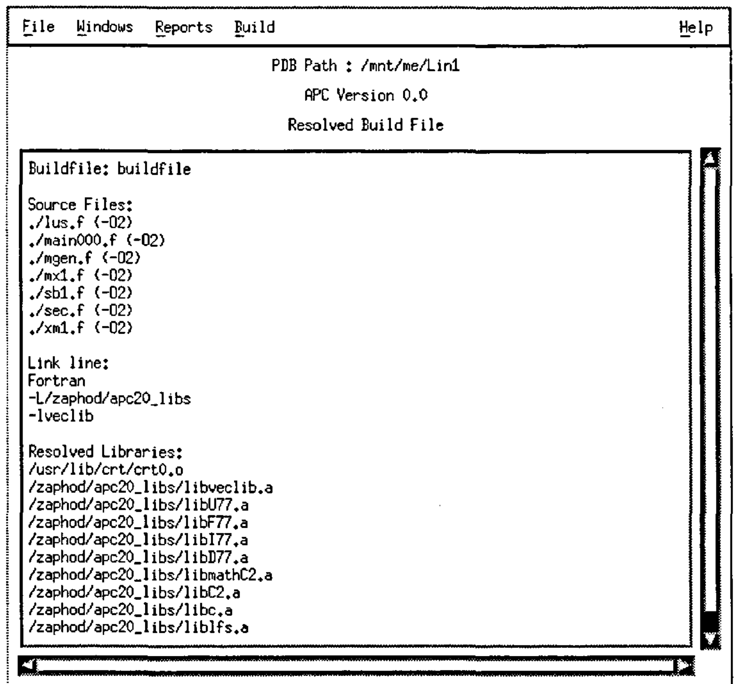
The Reports menu

The pane of the Main window can display the resolved contents of the buildfile or a program outline in place of the IPO report. To see one of these other reports, choose the appropriate item from the Reports menu, which is shown below.

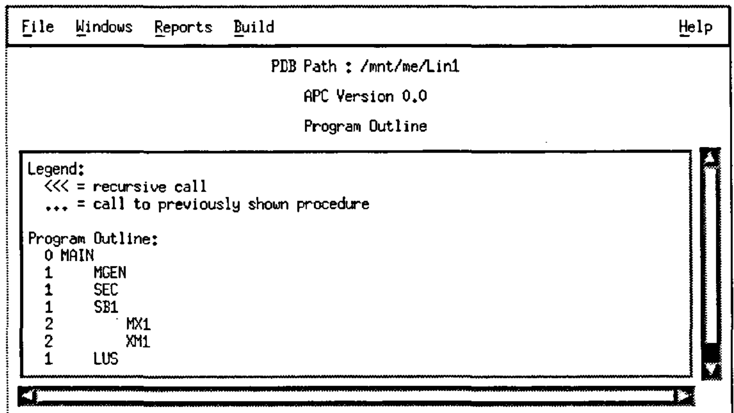


Since the reports are displayed in the main viewer window, you can only display one report at a time. The selection for the report currently displayed (in the above example, the IPO Report) is grayed out. If a report exceeds the available area of the pane, scroll bars appear, allowing you to move to any part of the report.

The Resolved Build File item displays an expanded form of the buildfile on which the current PDB is based. As shown in the following screen, this expanded buildfile lists each individual source file with its compilation options, each linked library with the loader option used to link it, and each individual library archive or object file name.



The Program Outline item displays an outline of the program which lists each procedure, indicating nested procedures by indentation, as shown below.



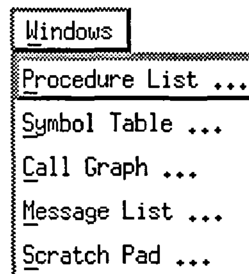
The Build menu

You can also build a new PDB from the Main window. To do this, select the Start item from the Build menu, which is shown below.

Selecting Start Build rebuilds the PDB using apc. Rebuilding the PDB opens an Xterm window with apc running in it. When the build is finished, a shell prompt will appear; typing "**exit**" here will dismiss the window.

The Windows menu

The Windows menu allows you to open other windows in addition to the Main window. These additional windows allow you to view more specific information about your program database concurrently with the summary information available in the Main window. The Windows menu is show below.



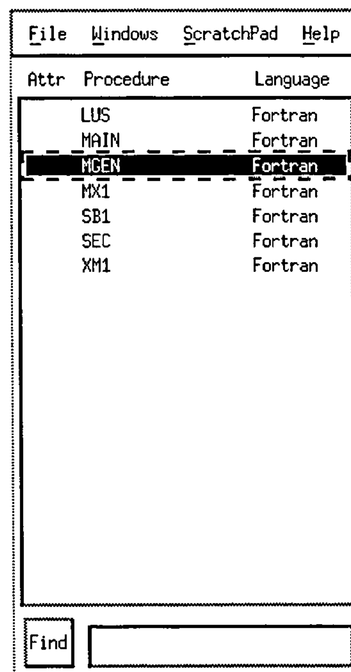
As shown, the Windows menu allows you to open a Procedure List, Symbol Table, Call Graph, Message List or Scratch Pad window. The Scratch Pad window is described in the previous section; each of the other windows is described in detail in a following section.

Procedure windows

Several windows are available to present information on procedures. The Procedure Description window provides detailed information on a single procedure. It can have different formats depending on whether you have access to the source for the procedure in question. The Call Information window, available from the Procedure Description window, provides further information on called procedures. The Procedure List window, available from the Windows menu of the Main window, allows you to chose a procedure to examine in a Procedure Description window.

The Procedure List window

The Procedure List window lists each procedure in your program database, along with its language and attribute information. When you choose Procedure List from the Windows menu in the Main window, a Procedure List window appears, as shown below.



Each procedure is listed in the Procedure column; the language it was written in appears under the Language column. The Attr column can display any of the following attributes for each procedure:

A: the procedure is contained in an APC library.

C: the procedure has clones.

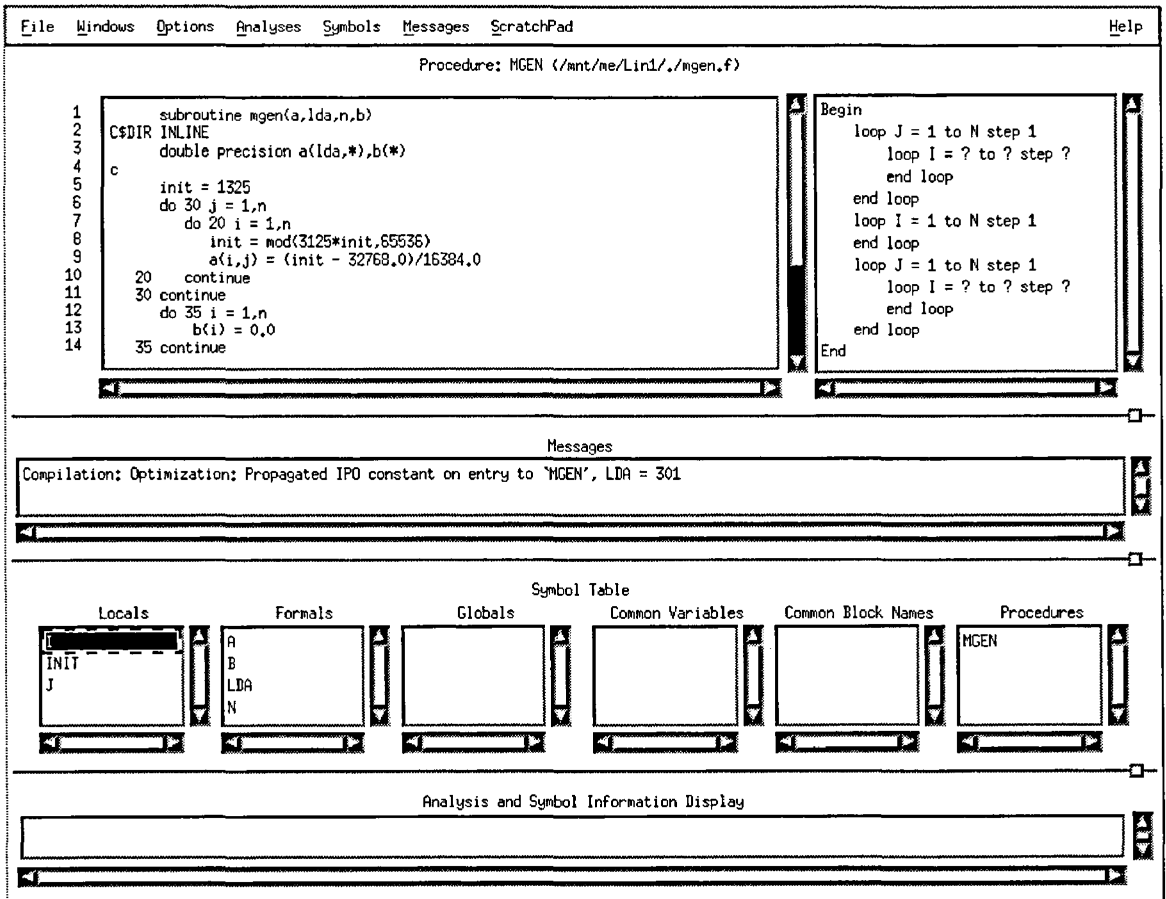
L: the procedure is contained in an annotated library.

U: the procedure is unreachable.

You can select a procedure from this list by clicking on it with the mouse, or you can enter its name into the text box in the lower right corner of the window and click the Find button to highlight the procedure (in the previous example, the procedure MGEN is highlighted). Command line wildcards can be used in the text box; clicking Find will then sequence through matches if there are more than one.

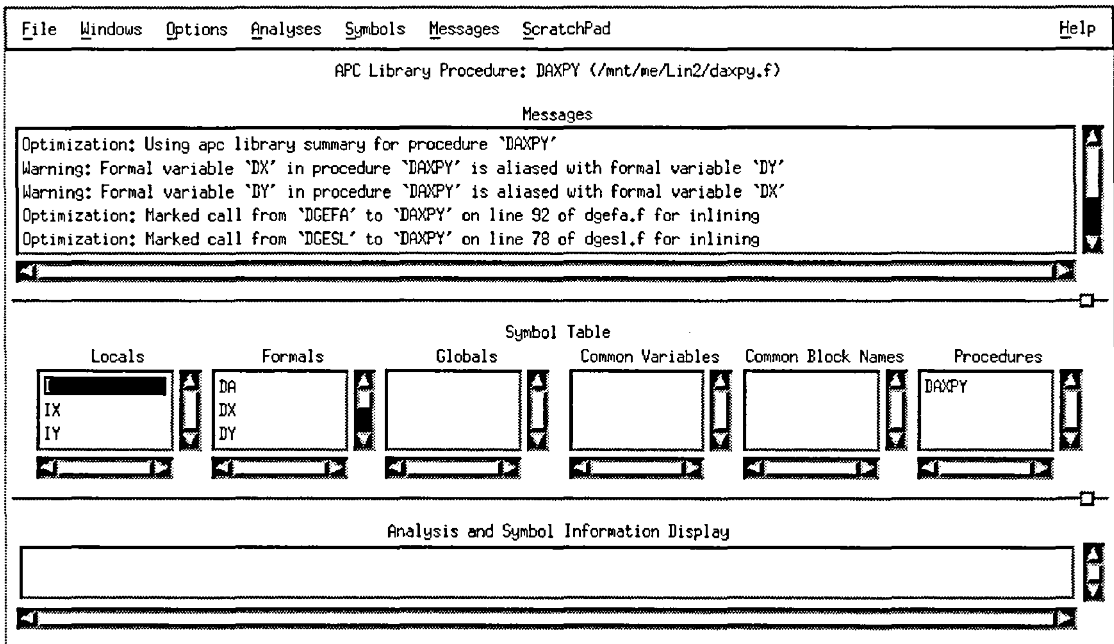
The Procedure Description window

Once a procedure is highlighted, you can see its Procedure Description window by choosing Procedure Description from the Windows menu or by pressing RETURN. You can also double-click on an unhighlighted procedure to bring up its Procedure Description window. The Procedure Description window for MGEN is shown below.



Procedure source section

The Procedure Description window is divided into four sections. The section titled "Procedure: *name*" provides side-by-side panes which contain the procedure's source code (on the left) and an outline of the procedure's control flow (on the right). These panes are linked together; clicking anywhere in the outline pane causes the source pane to scroll to the corresponding code. If you do not have access to the source code for the procedure, as with library routines, these panes are not available. An APC Library Procedure Description window is shown below.



Messages section

The Messages section lists error, warning and optimization messages generated by compiling the procedure. Clicking on a message will scroll the procedure source pane to display the appropriate source line.

You can control the number and type of messages displayed here by choosing Filter from the Messages menu. This will bring up the following window.

Classes

Fatal Error	Warning	Summary/Fault
Error	Advisory	Optimization/Repor

Set All | Clear All

Types

Array Subscripts	Pointers	Library Resolution
Common Blocks	Cloning	Renaming
Initializations	Inlining	Non-ansi FORTRAN 7
Types	Constants	Non-ansi FORTRAN 9
Aliases	Array Resizing	FORTRAN lint

Set All | Clear All

Regular Expression:

Matches regular expression

Does not match

OK | Apply | Cancel | Help

From this window you can not only specify the types and classes of messages displayed, you can select or reject those messages matching a certain regular expression.

To select or deselect message Classes or Types, click on the appropriate check boxes. When the box is filled in, it is selected. You can use the Set All and Clear All buttons to select or clear all check boxes at once. To use regular expressions, enter an expression (using grep-like expressions) in the text box and select either Matches regular expression or Does not match. When you click the Apply button, your Classes, Types, and, if selected, regular expression will be used to filter the Messages that appear.

Note

Class, Type and Regular Expression filters are applied concurrently. If no Class or Type is selected, or if Matches regular expression is selected but no expression is present, no matches will be found. Be sure to deselect both Regular Expression check boxes when you are not using Regular Expressions, and be sure that at least one Class or Type is selected.

Clicking the OK button closes the window, applying your selections and saving Class and Type settings for the next time you invoke the filter. Clicking the Cancel button closes the window without saving your current settings.

You can save the displayed messages by choosing Save... from the Messages menu. You will be prompted for an output file name.

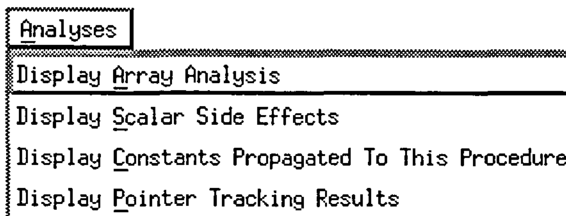
Symbol Table section

The Symbol Table section lists the procedure's symbols, displaying local variables, formal arguments, global variables, common variables, common blocks, and procedure names in separate panes. Single click on a symbol to select it for Symbol menu analysis, which is described in the following section.

Analysis and Symbol Information Display

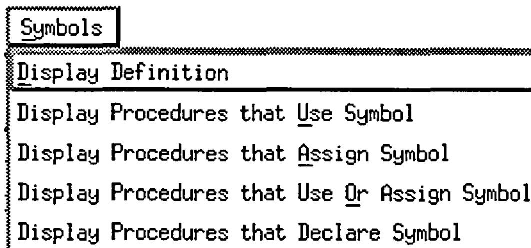
The Analysis and Symbol Information Display section displays analysis and symbol-definition data in response to your selections from the Analyses or Symbols menus.

You can select the analysis data to be displayed from the Analyses menu, which is shown below.



Selecting any item from this menu causes the appropriate analysis data to be appended to the current contents (if any) of the Analysis and Symbol Information Display pane.

The Symbols menu, which is available only when a symbol is selected in one of the Symbol Table panes, allows you to display various data on the selected symbol. This menu is shown below.



Selecting any item from this menu causes the appropriate symbol data to be appended to the current contents (if any) of the Analysis and Symbol Information Display pane.

Display control

Buttons appear on the right side of each section divider in the Procedure Description window, as shown below.

These buttons allow you to control the proportion of the screen occupied by each section. You can size the sections by clicking on and sliding these buttons vertically with the mouse.

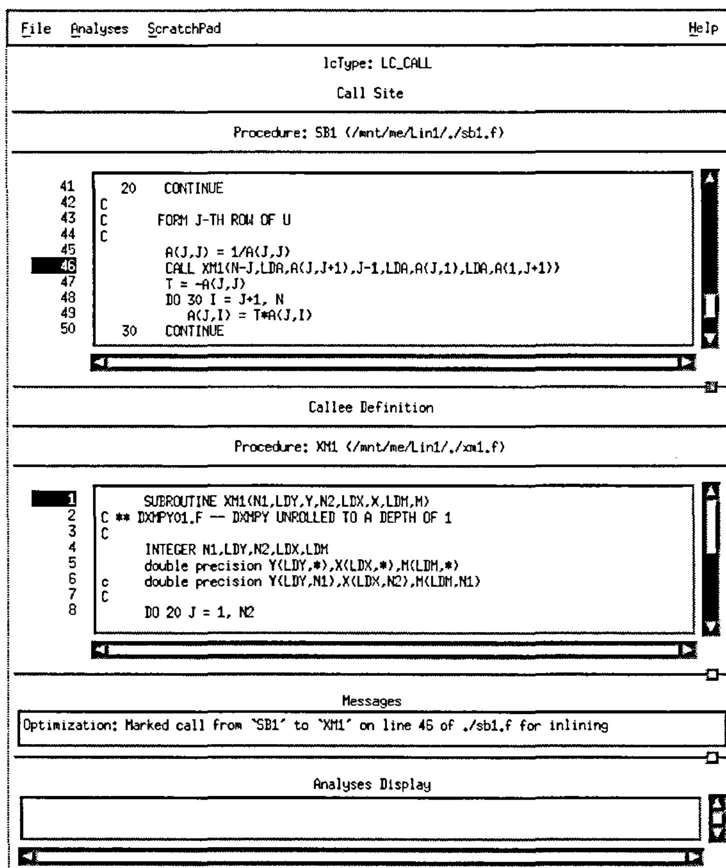
You can also completely hide the Messages and Symbol Table sections from view via the Options menu. This is a two-level menu; clicking on either of its items causes a subordinate menu to appear, as shown below.

Here, the Show item has been selected. Selecting the Hide item produces an identical submenu. To remove a section from the display, choose it from the Hide submenu. To bring it back, choose it from the Show submenu.

The Call Information window

The Windows menu of the Procedure Description window allows you to open a Call Information window, which displays information on a call made from the current procedure.

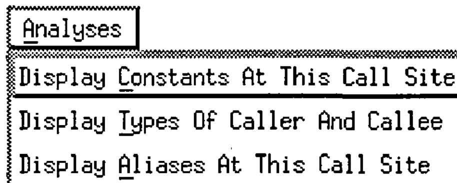
To open a Call Information window, click on a procedure call in the outline pane to select it and choose Call Information... from the Windows menu. An example Call Information window is shown below.



This window has two source-code panes: the Call Site pane, which shows the source of the calling procedure, and the Callee Definition pane, which shows the source code of the procedure being called.

The Messages pane shows any error, warning, or optimization messages associated with the call. Most calls do not generate a large number of messages, so this window does not have a menu item for Filter Messages like the one in the Procedure Description window.

The Analyses Display pane displays analyses you select from the Analyses menu, which is shown below.



Selecting any item from this menu causes the appropriate analysis data to be appended to the current contents (if any) of the Analysis Display pane.

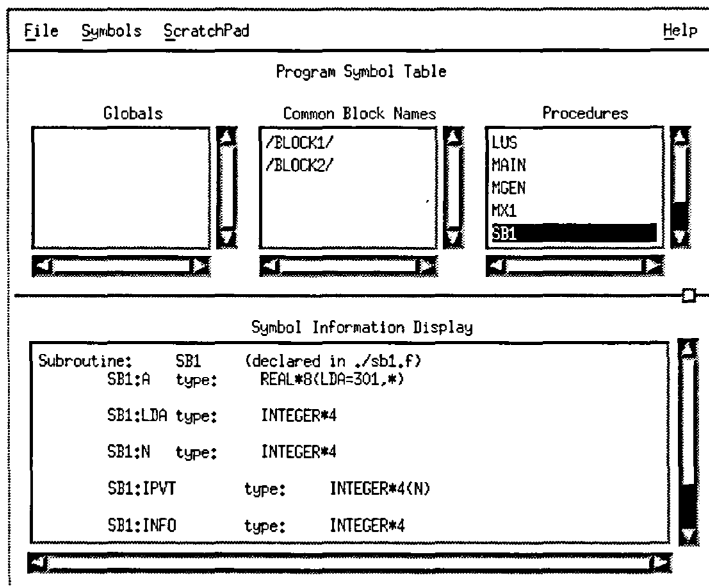
You can vertically size the Call window sections in the same manner as the Procedure Description window sections.

Editing the procedure source

The Windows menu of the Procedure Description window also allows you to edit the source to the current procedure if it is available. Choosing Edit Procedure from the Windows menu will bring up your default editor (specified in your `$EDITOR` environment variable) in a new window, with the current procedure loaded. Quitting the editor will kill the window.

The Symbol Table window

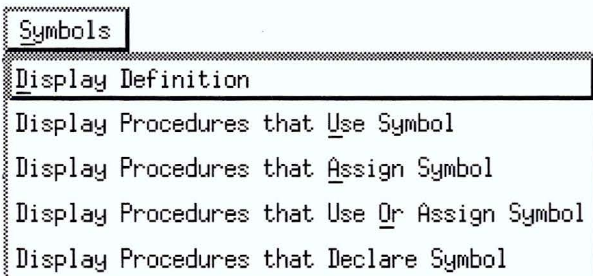
The Windows menu of the Main window allows you to open a Symbol Table window, which allows you to look at symbols defined throughout your program. To open the Symbol Table window, choose Symbol Table from the Windows menu. A Symbol Table window is shown below.



The Symbol Table window typically contains four panes: the Globals pane, which displays your program's global variables, the Common Block Names pane, which displays your program's common blocks, the Procedures pane, which displays all the procedures in your program database, and the Symbol Information Display pane, which displays information on the symbols listed in the other two panes based on selections you make from the Symbols menu. The Common Block Names pane is not present if your program contains no common blocks.

The Symbol Table window illustrated above is displaying definition information for the selected procedure `SB1`. This information was selected using the Display Definition item of the Symbols menu.

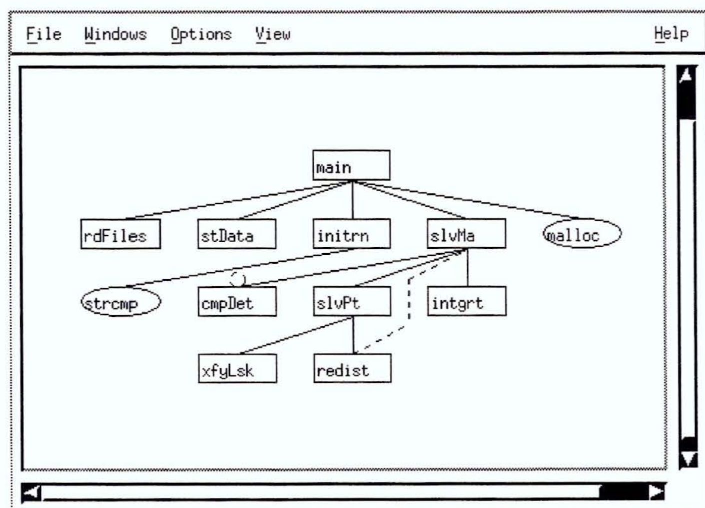
In addition to definition data, the Symbols menu allows you to view data on the use, assignment and declaration of the selected symbol, as shown below.



Information selected from the Symbols menu is appended to the Symbol Information Display.

The Call Graph window

The Call Graph window shows you a graphical representation of the calling hierarchy of your program. You can open a Call Graph window from the Main window by choosing Call Graph from the Windows menu. The Call Graph window is shown below.

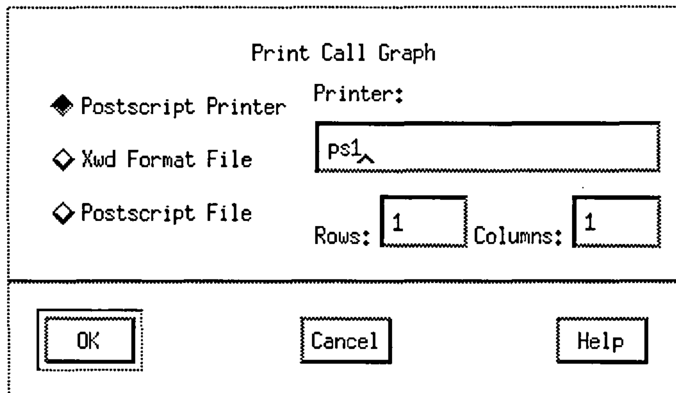


The call graph displays normal procedures as boxes and library procedures, like `malloc` and `strcmp` above, as ovals. It uses solid lines to indicate normal calls and dashed lines, like the one leading from `redist` to `slvMa`, to indicate indirect recursive calls. Dashed loops, such as the one attached to the procedure `cmpDet`, indicate immediate recursion.

The File menu

The Call Graph window's File menu allows you to print the graph as well as close the window.

Selecting Print Graph... produces the dialog shown below.



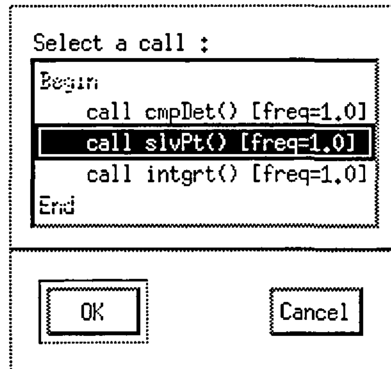
You can print the graph directly to the specified postscript printer, as shown in the dialog above, or to an xwd or postscript format file. Clicking on either file format choice changes the Printer: text box into a Filename: textbox into which you can enter your desired graph file name. The Rows: and Columns: boxes allow you to specify the number of vertical (rows) and horizontal (columns) pages the graph is to occupy.

The Windows menu

The Windows menu allows you to bring up a Procedure Description or Call Information window for any selected procedure in your Call Graph. To select a procedure, single click on it.

The Procedure Description window is described in the Procedure windows section of this appendix.

Selecting the Call Information window produces a Call Selection window, which is shown below.



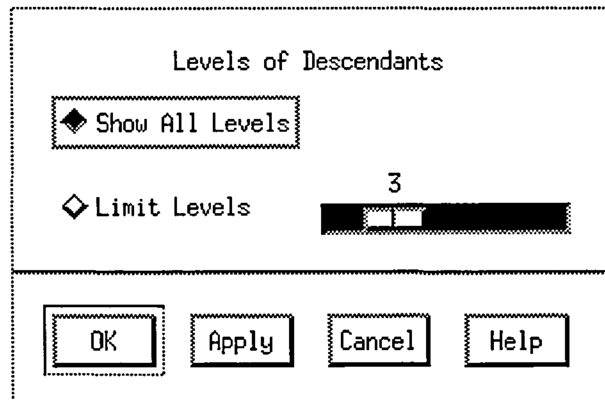
This window provides a procedure outline, with all lines grayed out except those that contain a procedure call. Select a procedure call for the procedure you want to see call information on and click OK or press RETURN. In the window above, the call to procedure `slvPt()` is selected.

The Call Information window that appears is described in detail in the Procedure windows section of this appendix.

The Options menu

The Options menu allows you to specify the levels of calls displayed in the graph, and to annotate the graphs with various data.

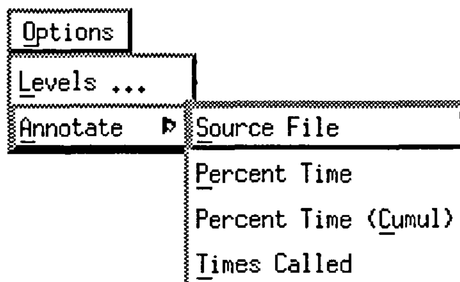
To specify the levels of calls, choose the Levels... item from the Options menu. This produces the window shown below.



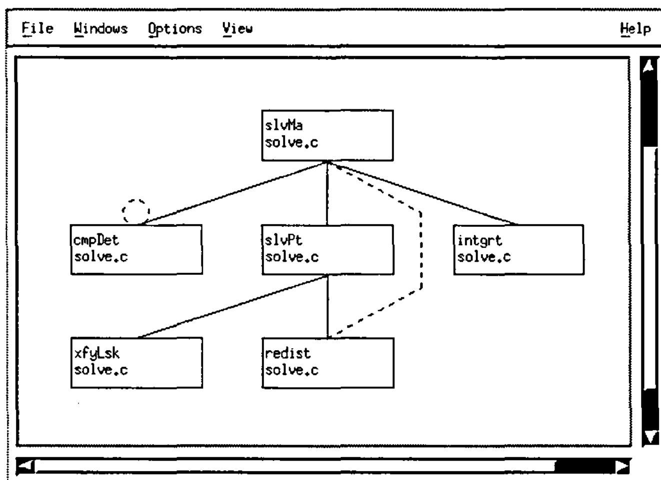
By default, Show All Levels is selected. To limit the levels shown, choose Limit Levels and adjust the slide to the desired number of

levels. Clicking OK will then apply your selection and kill the window; Apply will apply your selection without killing the window; Cancel will kill the window without applying the selection.

Choosing the Annotate item produces a submenu listing the various annotations that can be added to the graph nodes, as shown below.



By default, each node displays the procedure name and the source file in which it is contained. If this data is not visible at the default graph size, you can zoom in as described in the View menu section. The following graph section has been enlarged to show the full default contents of each node.



Choosing Percent Time from the annotate submenu annotates the nodes with the percentage of execution time the program spends in that node's procedure, excluding time spent in any called procedures. Percent Time (Cumul) annotates each node with the percentage of time spent in that procedure, including the time spent in all called procedures. Both Percent Time selections

require the availability of profile data (the procedures must be compiled with the appropriate options and profiled as described in Chapter 3, "Using the Application Compiler"). The Times Called item annotates each node with the number of times its procedure is called; this number is exact if the procedure has been profiled, but estimated if not.

The View menu

The View menu allows you to adjust various display aspects of the Call Graph. You can highlight procedures with specific attributes by selecting Highlight... from the View menu. This produces the Highlight Select window shown below.

Highlight: Times Called:

◇ Ancestors of procedure ◇ != 5

◇ Descendants of procedure ◇ <

◇ Source File is ◇ <=

◇ Percent Time ◆ =

◇ Percent Time (cumulative) ◇ >=

◆ Times Called ◇ >

◇ Uses variable

◇ Assigns variable

◇ Uses or assigns variable

◇ Declares variable

OK Apply Cancel Help

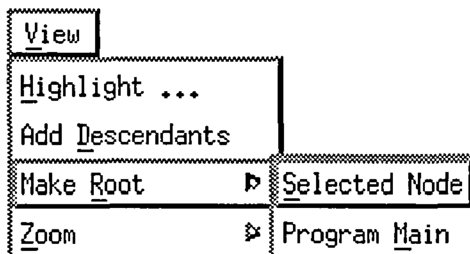
Pick the attribute you wish to use to highlight procedures by single-clicking a check box in the left column. In the above example, Times Called has been selected. Numeric selections like this will enable the right column of check boxes, which are otherwise disabled. For such selections, click on the desired operator from the second column and enter a value into the text box. In the above example, only those procedures that were called exactly 5 times would be highlighted.

The text box is used with all the highlight choices; when it is not used to hold a value, as shown above, it is used to hold a file name, procedure name, or variable name.

Clicking OK in this window applies your highlighting selections and kills the window; clicking Apply applies the selections and leaves the window active; clicking Cancel kills the window without applying your selection.

The View menu also allows you to regraph descendents of a specific procedure which extend below the descendant level limit specified via the Levels... item of the Options menu. To do this, select the procedure and choose Add Descendants from the View menu. The Add Descendants menu item is not available if the selected procedure has no descendants.

You can specify a procedure to use as the root of the graph by using the View menu's Make Root item. Make Root contains a submenu as shown below.

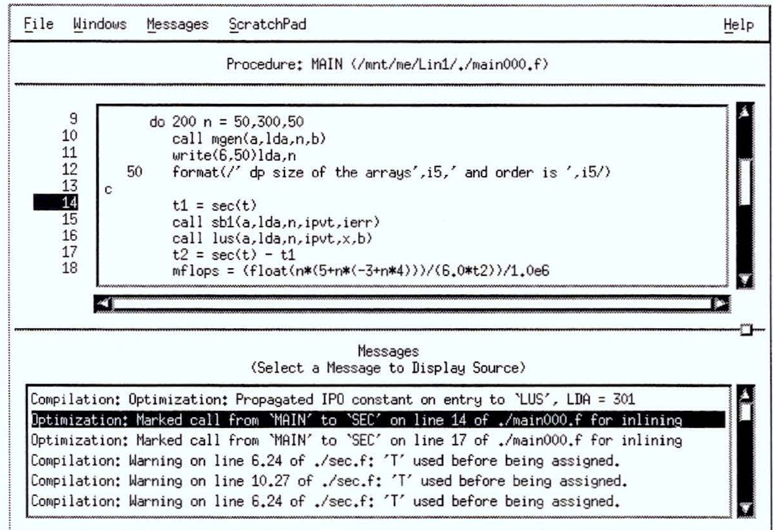


Choosing Selected Node will cause the currently selected node to become the root node; choosing Program Main will cause the Main node to become the root, which is the default condition.

You can enlarge or shrink the size of the graph by using the View menu's Zoom item. Choose Zoom Out from the Zoom submenu to make the graph smaller, or Zoom In to make it larger. It may be necessary to zoom in to make node information such as source file and procedure names completely visible in the graph.

The Messages window

The Messages window allows you to view all error, warning, and optimization messages generated by your program in a single window. You can open the Messages window from the Main window by choosing Message List... from the Windows menu. The Messages window is shown below.



The Messages window has two panes, a Procedure pane which displays the source code from the selected message, and a Messages pane which displays all the messages generated by your program. Clicking on a message in the Messages pane causes the Procedure pane to display the section of source code corresponding to it. The line number of the line of code corresponding to the selected message is highlighted centered in the pane. The procedure's name and source file path is displayed in the Procedure pane title. In the example above, an Optimization advisory is selected in the Messages pane, and, as you can see from the Procedure pane, the corresponding source line is line number 14 in procedure MAIN, which is contained in the source file /mnt/me/Lin1/main000.f.

You can filter the contents of the Messages pane using the same filter window used for the Procedure Description message filter. To filter the messages, select the Filter... item from the Messages menu. The Message Filter window that appears is thoroughly described in the Procedure Description section of this appendix.

You can save messages to a file by selecting the Save... item from the Messages menu. You will be prompted for an output filename.

You can bring up a Procedure Description window for the procedure appearing in the Procedure pane by choosing the

Procedure Description... item from the Windows menu. Refer to the Procedure Description window section of this appendix for more information on this window.

X resources

The PDB Viewer supports a set of X resources which parallel its command line options. When you set these resources in your `.Xdefaults` file, they specify certain viewer defaults.

X resources have the general form:

resourcename: resourcevalue

Where *resourcename* is the name of the resource, and *resourcevalue* is its value. The two must be separated by a semicolon and whitespace.

The PDB Viewer uses the following resources.

`Pdbview*binDir:` *altdir*
Use the alternate viewer executable files located in the directory *altdir.* Defaults to the directory in which the `apc` executable is installed, typically `/usr/convex/bin`. *altdir* does not apply to `pdbview`, the viewer driver.

`Pdbview*checkFileTimes:` *logexp*
If *logexp* is `true`, check the source files to see if they have been modified since the previous build. *logexp* defaults to `false`.

`Pdbview*contextRows:` *rows*
Set the number of lines around the call site in the Call Information window to *rows*. Defaults to 10.

`Pdbview*graphlevels:` *levels*
Set the number of levels displayed in the call graph to *levels*. By default, all levels will be displayed.

`Pdbview*helpPath:` *helpdir*
Use the alternate viewer help files located in the directory *helpdir.* Defaults to `.../pdbviewlib/onlineHelp`, where `...` is the directory in which pdbview is installed.

`Pdbview*helpRequest:` *logexp*
If *logexp* is `true`, display an online list of options. *logexp* defaults to `false`.

`Pdbview*path:` *path*

Open the PDB contained in the directory specified in *path.*

`Pdbview*prefetch:` *type*

Open the initial main window specified in *type*. *type* can take the value `ipo_summary` (the default), `build_info`, or `outline`.

`Pdbview*showXWarnings:` *logexp*

If *logexp* is `true`, allows the X server to write warning messages to `stderr`. *logexp* defaults to `false`.

`Pdbview*sourceColumns:` *cols*

Set the number of columns of source displayed in the source pane of the Procedure Description window to *cols*. Defaults to 80.

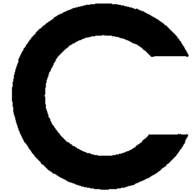
`Pdbview*sourceRows:` *rows*

Set the number of lines of source displayed in the source pane of the Procedure Description window to *rows*. Defaults to 20.

Geometry control resources are provided in `.../pdbviewlib/app-defaults/Pdbview`, where `...` is the directory in which the `pbdview` executable resides. Color resources are provided in `.../pdbviewlib/app-defaults/Pdbview-color`. To use these color resources, you must specify one of the following lines in the `.Xdefaults` file in your home directory:

```
*customization: -color
Pdbview*customization: -color
```

Error and warning messages



The Application Compiler produces messages and advisories that are not supported by the conventional compilers. This appendix describes some common problems that can produce these messages, and recommends steps that you can take to identify and correct such problems. In addition to the error messages described here, the Application Compiler can generate any of the error messages generated by the conventional compilers. Consult the *Fortran User's Guide* or *C User's Guide* for information on those messages.

Error messages

The Application Compiler generates an error message when some condition makes it impossible to finish compiling. Such an error may force the Application Compiler to terminate immediately.

Interprocedural compilation errors

```
Error: Unresolved procedure symbol '<procedure>'
```

The program calls a procedure that the Application Compiler cannot locate. Check the spelling of the procedure name. Check your buildfile to make sure that it specifies all the source and object files required to create your program, and check your directories to make sure the required files are in the right places. If the unresolved symbols are defined in a library that will be manually linked later, you can use the `-permit unresolved` option, as described in Chapter 3.

```
Error: Procedure '<procedure>' is addressed but
not defined
```

The Application Compiler cannot find a procedure whose address is referenced by your program. This may be the result of a spelling error, a missing source file, or a mistake in specifying directories in your buildfile.

```
Error: More than one 'main' entry point found in
<file>
```

The Application Compiler has found more than one `main` function or program `MAIN` section. It cannot know which one you want to use, so this is a fatal error. Usually, this error occurs because of old backup files containing duplicate main sections. Some text editors create these backup files automatically. Check your working directories for backup source files and delete them or move them to another directory.

```
Error: No 'main' entry point found
```

The Application Compiler did not find a `main` function or program `MAIN` section. The Application Compiler must compile the main procedure. You cannot link a previously compiled `main.o` into a new application.

```
Error: Function '<procedure>' multiply defined
in file '<file>' and file '<file>'
```

Your program defines the indicated procedure twice, in two separate source files. Make sure your directories contain only current versions of your source files, remove out-of-date source files, or modify your buildfile to remove unneeded sources.

```
Error: Invalid option '<option>'
```

You used an invalid option to `apc`. The Application Compiler stops compiling. Type `apc -help` to see a list of all valid options or consult Chapter 3.

Buildfile errors

```
Error: line <#> of <buildfile>, object files may
appear only on the link line
```

If you list an object file to be linked into your executable, it must be listed on the line that begins with the word `link`.

```
Error: line <#> of <buildfile>, may specify
either Fortran or C libraries, but not both
```

Your buildfile includes both Fortran and C on the link line. You cannot link both libraries. If your program uses both languages, link only with the Fortran library.

```
Error: line <#> of <buildfile>, source file
not allowed on link line
```

Your buildfile lists a source file on the link line. Source files must be listed on a separate line and must not be preceded by the keyword `link`.

```
Error: line <#> of <buildfile>, expected <token>
but saw <token>
```

The Application Compiler found an unexpected token while parsing your buildfile.

Variable problems

```
Error: Variable '<variable>' initialized in file
'<file>' and file '<file>'
```

A source file reinitializes a variable already initialized in another file. Check whether both files initialize the variable the same way. If so, delete one of the initializations. If not, you may need to create a new variable and use it for one of the initializations.

```
Error: Variable '<variable>' declared
inconsistently in file '<file>' and file
'<file>'
```

Two source files contain different declarations for the same variable. You may have accidentally given two variables the same name. If so, rename one of them. Otherwise, determine which declaration is correct and delete the other one.

Warning messages

Warning messages describe nonfatal conditions, which do not prevent the Application Compiler from producing an executable. A warning may indicate a potential problem in your code, or a potentially dangerous situation that arises from the way the program is compiled. In addition to the warning messages described here, the Application Compiler can generate the same warnings reported by the conventional compilers. For a description of the conventional warning messages, see the *Fortran User's Guide* or *C User's Guide.*

Argument problems

```
Warning: Argument number <#> of '<procedure>'
has inconsistent type in <file> on line <#>
and <file> on line <#>
```

The actual and formal argument types don't match. Check the calling procedure to make sure the correct arguments are being passed. Check the procedure being called to make sure the formal arguments have the correct type definitions. Code that does not comply with ANSI standards for type compatibility may produce correct answers, but still generate inconsistent-type messages.

```
Warning: Argument number <#> of '<procedure>'
has inconsistent type in <file> on line <#>
and <file> on line <#> due to compiler generated
formals.
```

For some formal parameter types, such as Fortran's `CHARACTER` type, the compiler generates extra parameters (for the `CHARACTER` type, the length is passes as a parameter). Incorrectly passing such parameters can cause this warning.

```
Warning: Argument number <#> of possible call to
'<procedure>' has inconsistent type in <file> on
line <#> and <file> on line <#>
```

The program makes an indirect call, which may be directed at runtime to the specified procedure. The actual argument passed does not match the type expected by the corresponding formal argument. Make sure the indirect call is not directed to the wrong procedure. Check the calling procedure to make sure the correct arguments are being passed. Check the procedure being called to make sure the formal arguments have the correct type definitions. Code that does not comply with ANSI standards for type compatibility may produce correct answers, but still generate inconsistent-type messages.

```
Warning: Call to `<procedure>' in `<file> on
line <#> has fewer arguments than definition
in <file> on line <#>
```

The call passes fewer actual arguments than the procedure expects.

Check the declaration of the called procedure. Make sure all necessary arguments are declared. If the declaration appears correct, look at the source code from which the call is made. Determine which actual argument is missing, and add it to the argument list.

```
Warning: Call to '<procedure>' in <file> on line
<#> has more arguments than definition in <file>
on line <#>
```

The call passes more actual arguments than the procedure expects. Compilation continues, and the extra arguments at the end of the argument list are ignored.

Check the declaration of the called procedure. Make sure all necessary arguments are declared. If the declaration appears correct, look at the source code from which the call is made. Determine which actual argument is unneeded, and delete it from the argument list.

```
Warning: Possible call to `<procedure>' in
<file> on line <#> has fewer arguments than
definition in <file> on line <#>
```

The program makes an indirect call, which may be directed at runtime to the specified procedure. The call passes fewer actual arguments than that particular procedure expects. Compilation continues, and the extra arguments at the end of the argument list are ignored.

Check the declaration of the called procedure. Make sure all necessary arguments are declared. If the declaration appears correct, look at the source code from which the call is made. Make sure the indirect call is not directed to the wrong procedure. If the procedure is indeed a valid target for that call, check the list of arguments being passed. Determine which of the actual arguments is missing, and add it to the argument list.

```
Warning: Possible call to '<procedure>' in
<file> on line <#> has more arguments than
definition in <file> on line <#>
```

The program makes an indirect call, which may be directed at runtime to the specified procedure. The call passes more actual arguments than that particular procedure expects. Because all formal arguments receive a value, the code may still run correctly, so this is a warning rather than an error. Extra arguments at the end of the argument list are ignored.

Check the declaration of the called procedure. Make sure all necessary arguments are declared. If the declaration appears correct, look at the source code from which the call is made. Make sure the indirect call is not directed to the wrong procedure. If the procedure is indeed a valid target for that call, check the list of arguments being passed. Determine which of the actual arguments is unneeded, and delete it from the argument list.

```
Warning: '<procedure>' calls '<procedure>'
passing constant actual to array formal
'<variable>'
```

The call passes a constant argument where the procedure expects an array. Check to make sure you are calling the correct procedure at that point in your program. If so, determine which array should be passed to produce the result your program expects.

```
Warning: '<procedure>' calls '<procedure>'
passing scalar actual '<variable>' to array
formal '<variable>'
```

Your code passes a scalar variable in an argument position where the procedure expects an array. Check to see if your code is passing only one element instead of the entire array. Check to make sure you are calling the correct procedure at that point in your program. If so, determine which array should be passed to produce the result your program expects.

```
Warning: Unused argument '<variable>' in
procedure '<procedure>'
```

A formal argument is defined in the argument list of a procedure but not used in that procedure. Check the code for the procedure for missing statements. If you do not find any missing statements, and the procedure's logic is correct, the argument is not needed. Delete it from the procedure's argument list and adjust all calls to the procedure accordingly.

Definition problems

```
Warning: Procedure '<procedure>' is defined
but not called
```

Your program never calls the indicated procedure. The Application Compiler can still produce an executable program, so this is only a warning. It may, however, indicate a logic error in your program. Check your source code for a missing call statement. Also check your source directory to make sure there are no source files that don't belong there. Remove any unneeded source files from your directory or buildfile.

You may notice the Application Compiler still compiles procedures that are flagged as "not called." This happens when a procedure contains global initializations that must be compiled even if the procedure is not called. In Fortran, for example, a procedure that is never called can initialize COMMON-block variables. In C, a function can also contain global-variable definitions. Like variable initializations, these definitions must be

compiled even if the procedure is never called. For this reason, the Application Compiler always compiles C functions, whether they are called or not. To reduce compile time, however, the Application Compiler only compiles the data, not the code statements, for procedures that are never called.

Pointer problems

`Warning: Pointer '<variable>' does not contain a valid address on entry to '<procedure>'`

The pointer has never been assigned an address, or it was assigned an invalid address. Determine the procedure and code section where the pointer should have been assigned. Make sure the assignment is correct. If you find an assignment that appears correct, look for additional, accidental assignments that occur later on.

`Warning: Pointer '<variable>' was never assigned a valid address`

A pointer variable is used without having been assigned an address. Look for a missing assignment statement.

`Warning: Expected pointer return on line <#>`

The function definition indicates that a function returns a pointer, but the value returned is not a pointer. Check that the function's declaration is correct. Should it really return a pointer type? Check the function's `return` statement to make sure the correct value is returned.

`Warning: Invalid RHS of assign to pointer '<variable>' on line <#>`

The right-hand side of an assignment statement does not produce a valid address value. Make sure the correct variable appears on the left-hand side of the assignment statement. Make sure the right-hand side follows the proper rules for pointer arithmetic and uses only variables of the correct type.

Return-value problems

`Warning: Return value of call to '<procedure>' has inconsistent type in <file> on line <#> and <file> on line <#>`

The called procedure returns a type different from what the call expects. Check the function's declared type; if it is correct, make sure that the calling procedure expects the correct type.

```
Warning: Return value of possible call to
'<procedure>' has inconsistent type in <file> on
line <#> and <file> on line <#>
```

The program makes an indirect call, which may be to the procedure specified. The procedure returns a type different from what the call expects. Check the procedure's declared type; if it is correct, make sure that the calling procedure expects the correct type.

Procedure problems

```
Warning: Assuming default information for
unannotated procedure '<procedure>'
```

The Application Compiler cannot find annotations for the specified procedure. Optimization proceeds using default assumptions that the unannotated procedure can modify any argument or global variable and alias any pointer. Interprocedural optimization is degraded.

```
Warning: Calls to ``<procedure>'' library
procedure can cause incorrect results with the
Application Compiler, especially at higher
optimization levels
```

Calls to signal, setjmp, longjmp or sigvec can create side effects that prevent the APC from correctly optimizing code. If any of these routines is called from your program, this warning is issued.

Assignment problems

```
Assignment(s) to variable ``<variable>'' in
procedure ``<procedure>'' may cause incorrect
results due to alias with variable `<variable>'
```

This warning is issued when a hidden alias exists between two actual parameters in a procedure. Refer to Chapter 2, "Basic interprocedural optimizations", for help recognizing hidden aliases.

Other warnings

```
Warning on line <#> of <file>: '<variable>'
may be used before being assigned.
```

This warning means that at least one path exists which, if executed, would result in the variable being unassigned. The

compiler cannot tell whether that path will be taken at runtime. Examine the control flow of your program to find the path which results in the use of an unassigned variable. If the path cannot be taken, it is dead code; remove it from your program. Otherwise, add an assignment for the variable to avoid a possible logic error.

```
Warning: Uses of array '<variable>' may have
subscripts less than lower bound in
'<procedure>'
```

Depending on what happens at runtime, your code may use the value of an element that is located outside the declared bounds of the indicated array. Examine your code to determine whether this can actually happen, and repair the problem if necessary.

```
Warning: Assigns to array '<variable>' may
have subscripts less than lower bound in
'<procedure>'
```

Depending on what happens at runtime, your code may assign a value to an element that is located outside the declared bounds of the indicated array. Examine your code to determine whether this can actually happen, and repair the problem if necessary.

```
Warning: Uses of array '<variable>' may have
subscripts greater than upper bound in
'<procedure>'
```

Depending on what happens at runtime, your code may use the value of an element that is located outside the declared bounds of the indicated array. Examine your code to determine whether this can actually happen, and repair the problem if necessary.

```
Warning: Assigns to array '<variable>' may
have subscripts greater than upper bound in
'<procedure>'
```

Depending on what happens at runtime, your code may assign a value to an element that is located outside the declared bounds of the indicated array. Examine your code to determine whether this can actually happen, and repair the problem if necessary.

```
Warning: Formal variable '<variable>' in
procedure '<procedure>' is aliased with global
variable '<variable>'
```

The passing of an actual argument causes the formal argument to become aliased with a global. This can result in wrong answers when the global or formal is assigned. Rewrite your code to eliminate the alias. Use a `no_alias` directive to suppress this message if no aliasing problem actually exists.

```
Warning: Formal variable '<variable>' in
procedure '<procedure>' is aliased with formal
variable '<variable>'
```

The same pointer or array is passed to two formal arguments in the same procedure, thus creating an alias. This is legal in C, but may be accidental. The ANSI Fortran standard prohibits this practice. In either language, it can lead to aliasing problems that reduce the optimization of your program. Examine your code to determine if an aliasing problem actually exists. If it does not, use a `no_alias` statement in your buildfile to suppress this warning.

```
Warning: '<procedure>' uses '<variable>',
which is neither assigned nor initialized
```

The procedure uses an uninitialized variable. Often, this is the result of a Fortran program assuming that COMMON-block variables are automatically initialized to zero. The ANSI Fortran standard does not support this assumption. See Chapter 8, "Fortran hints", for additional guidance.

```
Warning: Assignment to '<variable>' in
'<procedure>' is invalid due to alias with
variable '<variable>'
```

The ANSI Fortran Standard states that dummy arguments and COMMON variables must not be aliased with one another if either is assigned. Most Fortran compilers cannot detect this error. Your code assigns a value to an aliased variable, thus changing the value of another variable. Rewrite your code to eliminate the alias.

Glossary

A term in *italic* indicates a cross reference to another item in the glossary.

A

actual argument
In Fortran, a value that is passed by a call to a procedure (function or subroutine). The actual argument appears in the source of the calling procedure; the argument that appears in the source of the called procedure is a *dummy argument.* C conventions refer to actual arguments as *actual parameters.*

actual parameter
In C, a value that is passed by a call to a procedure (function). The actual parameter appears in the source of the calling procedure; the parameter that appears in the source of the called procedure is a *formal parameter.* Fortran conventions refer to actual parameters as *actual arguments.*

alias
An alternative name for some object, especially an alternative variable name that refers to a memory location. Aliases can cause recurrences, which prevent the compiler from vectorizing or parallelizing parts of a program.

analysis phase
The forth phase of the interprocedural compilation process, during which the Application Compiler performs a second scalar optimization analysis, using information generated in part 1 of the synthesis phase and storing its results to the program database.

annotations
Psum directives used to provide the APC with optimization information for procedures for which you do not have access to the source. Annotations are stored in an annotation file that is associated with the object or library file to which they apply. See *psum directives.*

Glossary

APC libraries
Special precompiled libraries that are compiled (and optimized) by the Application Compiler. These libraries offer higher performance than non-APC libraries, and speed compile times when linked with APC-compiled applications.

`apc` command
The user-visible interface to the Application Compiler. The `apc` command reads specifications from a buildfile and calls several other programs to create an executable program.

apparent recurrence
A condition or construct that fails to provide the compiler with sufficient information to determine whether or not a recurrence exists. Also called a *potential recurrence.*

argument
In Fortran, either a variable declared in the argument list of a procedure (function or subroutine) that receives a value when the procedure is called (*dummy argument*) or the variable or constant that is passed by a call to a procedure (*actual argument*). C conventions refer to arguments as *parameters.*

B

basic block
A linear sequence of program statements with a single entry and a single exit.

buildfile
A text file, analogous to a make file in function but not in structure or in content, that contains specifications for the `apc` command controlling the compilation of a program.

`build` utility
What the `apc` command was called prior to Version 2.1 of the Application Compiler. See *`apc` command.*

C

cloning
See "procedure cloning."

compile phase
The second to last phase of the interprocedural compilation process, which creates the optimized object code.

constant propagation
The replacement of variable references, by the compiler, with a constant value previously assigned to that variable, performed within a single procedure by conventional compilers and between procedures by the Application Compiler.

conventional compiler
A compiler that cannot perform interprocedural optimization. This term encompasses all Convex compilers except the Application Compiler and virtually all compilers available from other vendors.

D

data localization
Optimizations designed to keep frequently used data in processor local memory. On C Series machines, data localization is accomplished using the vector registers; on SPP Series machines, it is accomplished using the data cache.

data privatization
The process of insuring that none of a procedure's local data contains a value between two calls to the procedure. This is necessary in order to call the procedure in parallel.

database, program
A set of files created and maintained by the Application Compiler to store the intermediate representation code and optimization information about the program being compiled.

compiler directive
A means of inserting supplemental optimization information into program source code. Compiler directives look like comments to compilers that are not designed to recognize them.

dummy argument
In Fortran, a variable declared in the argument list of a procedure (function or subroutine) that receives a value when the procedure is called. The dummy argument appears in the source of the called procedure; the parameter that appears in the source of the calling procedure is an *actual argument*. C conventions refer to dummy arguments as formal parameters.

F

formal parameter
In C, a variable declared in the parameter list of a procedure (function) that receives a value when the procedure is called. The formal parameter appears in the source of the called procedure; the parameter that appears in the source of the calling procedure is an *actual parameter*. Fortran conventions refer to formal parameters as dummy arguments.

function
A procedure that returns a value. Any procedure in C or a procedure defined as a `FUNCTION` in Fortran.

Glossary

G

global variable

A variable whose scope is the entire program. In C programs, a global variable is a variable that is defined outside of any one procedure. In Fortran, it is any variable declared in a `COMMON` block.

global optimization

A restructuring of program statements that is not confined to a single basic block. Global optimization, unlike interprocedural optimization, is confined to a single procedure. Global optimization is done by all Convex compilers at optimization level `-O1` and above

H

hidden alias

An alias that, because of the structure of a program or the standards of the language, goes undetected by the compiler. Hidden aliases can result in undetected *recurrences*, which may result in wrong answers.

I

inlining

The replacement of a procedure (function or subroutine) call, within the source of a calling procedure, by a copy of the called procedure's code.

interprocedural optimization

Restructuring of program statements, not confined in scope to a single procedure, to improve performance.

L

link phase

The seventh phase of the interprocedural compilation process, during which the Application Compiler invokes the linker to link APC-compiled object files with any libraries that are used in the program to produce an executable.

local optimization

Restructuring of program statements within the scope of a basic block. Local optimization is done by all Convex compilers at optimization level `-O0` and above.

M

`MAIN` program

In a Fortran program, the program section invoked by the operating system when the program starts up and containing program statements that are not part of any function or subroutine.

main procedure
A procedure invoked by the operating system when the program starts up. The main procedure is the program `MAIN` section in Fortran; in C, it is the function `main()`.

memory bank conflict
An attempt to access a particular memory bank before a previous access to the bank is complete.

P

parallel optimization
The transformation of source into parallel code (parallelization) and restructuring of code to enhance parallel performance.

parameter
In C, either a variable declared in the parameter list of a procedure (function) that receives a value when the procedure is called (*formal parameter*) or the variable or constant that is passed by a call to a procedure (*actual parameter*). Fortran conventions refer to parameters as *arguments.*

pdbview utility
A graphical interface to the data contained in the program database.

pointer tracking
A powerful aliasing algorithm used by the Application Compiler, which tracks the memory locations to which each pointer can be set.

potential recurrence
A condition or construct that fails to provide the compiler with sufficient information to determine whether or not a recurrence exists. Also called an *apparent recurrence.*

procedure
A self-contained unit of code that is called by another procedure or, in the case of a *main procedure,* invoked by the operating system when the program starts up. A procedure can be a subroutine, function, or program `MAIN` section in Fortran, or a function in C.

procedure cloning
The creation of duplicate procedures (clones) that enable the Application Compiler to perform different optimizations for different calls to the same procedure.

program database
A set of files created and maintained by the Application Compiler to store the intermediate representation code and optimization information about the program being compiled.

psum (procedure summary) directives
Special compiler directives used to annotate procedures with behavioral information needed by the Application Compiler. psum directives are most useful when linking object or assembly language files for which you do not have the source; in this case, the information they provide can help the Application Compiler to optimize your source code which calls procedures contained in the object or assembler files.

pure procedure
A procedure that doesn't depend upon or change the state of global variables.

R

recurrence
A condition or construct in which data created on one iteration of a loop is referenced on a subsequent iteration.

S

scalar optimization
The restructuring of program statements to improve performance, not including transformations that produce parallel or vector code.

self-dependence
A condition in which a procedure contains local static arrays and/or scalars that are assigned in one call to the procedure and referenced in a later call.

source dependency
Data describing the dependencies or relationships between source files that require compiles to be performed in a specific order. The Application Compiler obtains dependency information by scanning the target application's source files.

source file
An ASCII text file, created by the programmer, that is compiled to produce executable code.

strip mining
The creation of an outer loop to allow the compiler to vectorize an original loop with a length greater than, or possibly greater than, the vector-register length of 128.

subroutine
In a Fortran program, any procedure that is not a function or the program's `MAIN` section.

summary phase
The phase of the interprocedural compilation process during

which the Application Compiler analyzes each procedure, generates an intermediate-code representation for it, performs scalar optimizations on it, and stores it in the program database.

synthesis phase, part 1
The phase of the interprocedural compilation process during which the Application Compiler gathers optimization information for pointers, procedure calls, and variables that are passed by reference, and stores it in the program database.

synthesis phase, part 2
The phase of the interprocedural compilation process during which the Application Compiler performs interprocedural optimization analysis and gathers information on procedure side effects, storing its results in the program database.

V

vector optimization
The transformation of loops into vector code (vectorization) and restructuring of code to enhance vector performance.

Index

Symbols

#define 170

A

-al option 167
actual argument
  defined 211
actual parameter
  defined 211
adding parallelization 158
adjustable arrays 165
algorithms
  aliasing 48
aliases 46, 48, 49, 51
  defined 46, 211
  hidden 12, 15
  invalid 97
  potential 48
aliasing algorithms 48
analysis
  defined 211
-anno procedural compiler option 119
anno.map file 106
anno_ar utility 127
  examples 128
annotate buildfile statement 105
annotated libraries 107
annotations
  argument declarations 118
  example 125
  simple example 118
  specifying mappings 105
ANSI aliasing 48
ANSI FORTRAN standard 161, 166
ANSI FORTRAN standard checking 112
-ansi77 compiler option 112
-ansi90 compiler option 112
apc
  invoking 53
apc command
  defined 212
apc command 6, 53, 55
APC libraries 116
apc options 55, 82
apparent recurrence 14
  defined 212
argument
  defined 212
arguments
  array 45
  dummy 25, 51
  incompatible 45
  scalar 45
  too few 45, 97
  too many 45, 97
array arguments 45
array bounds 45
array-storage optimization 84
ASCII text editors 54
ASCII text files 99
assembly-language files 117
-assert option 84
-assert subscripts_ok option 52
assistance
  calling TAC xviii
assumed-size arrays 165
automatic initialization 161

B

-B compiler option 172
backup source files 109
bank conflicts 51, 84
basic block 17
  defined 5, 212
benefits of optimization 12
bibliography xvi
binary search 153, 155, 158
binding 163
  dynamic 57
BLOCK DATA 163
bounds
  array 45
buildfile
  defined 212
  force_object command 107
buildfiles 6, 53, 99
  flag macros 109
  optimization statements 100
  specifying directories in 108
  var_args statement 104

C

C and FORTRAN
  combining 113

-c compiler option 100, 172
C libraries 100
C library functions 169
C macros 170
-c option 89, 159
C source files 99
cache misses 51
calling C from FORTRAN 113
calling FORTRAN from C 113
calls
  embedded 14
-cfc compiler option 106
CFLAGS macro 110
char arguments 141
CHARACTER
  arguments 71, 141
CHARACTER variables 166
-check arrays option 166, 167
-check common option 59
-check option 58, 97
-check options 55
-check types option 63, 78
-chk compiler option 112
-clone all option 84, 112
CLONE directive 84, 139, 140, 141, 142, 143
-clone none option 84, 112
-clone option 84
clone statement 100
clones
  reusing 32
cloning 28, 29, 30, 32, 54, 69, 75, 95, 97, 113
  defined 212
  directives 139
combining C and FORTRAN 113
comments
  buildfile 99
COMMON-block variables 161
  propagating 22
-compat rff=new option 171
compilation dependencies 54, 89
compile phase
  defined 212
compiler option
  -ansi77 112
  -ansi90 112
  -chk 112
compiler options 54, 171
  -B 172
  -c 100, 172
  -cfc 106
  -D 172
  -db 106, 153
  -ds 156
  -I 111
  -il 100
  -is 100
  -o 172
  -p 106, 155, 158, 159
  -pa 106, 155, 158, 159
  -pcc 106
  -pg 106
  -S 172
  -vfc 106
-compress option 89
CONDITION_TRUE directive 146, 147
conditionals 156
conflicts, bank 51
constant propagation 17, 18, 19, 20, 21, 22, 24, 25, 28, 70, 97
  defined 212
constants
  Hollerith 166
-cont option 88
conventional compiler
  defined 213
conventional optimization 53
conventions
  notational xv
Convex Performance Analyzer 91, 154
Cray-style pointers 168
creating buildfiles 53, 99
creating psum files 115
csd debugger 153
CXpa performance analyzer 91
CXpa profiler 154

D

data independence 13
database
  program, defined 213
-db compiler option 106, 153
dead code
  eliminating 27
debugger
  csd 153
debugging 53, 99, 153
declarations
  multiple 45
default optimization level 2
dependencies
  compilation 54, 89
directive
  CLONE 139, 140, 141, 142, 143
  CONDITION_TRUE 146, 147
  ESTIMATED_TRIPS 120, 124, 125, 147, 148, 149, 150
  FORCE_PARALLEL 14
  INLINE 131, 134, 143
  INLINE_CALL 134, 135, 138
  NO_CLONE 144, 145
  NO_INLINE 94, 133, 135, 145
  NO_INLINE_CALL 136, 137, 138

NO_RECURRENCE 14
PSUM_ASGS 121
PSUM_KILLS 121
psum_range_flags 123
psum_range_names 122, 124
PSUM_REENTRANT 124
PSUM_USES 122
scalar 156
directives 131
informational 146
inlining 131
directories 108
documentation
ordering xvii
-ds compiler option 156
dummy argument
defined 213
dummy arguments 25, 51
duplicate procedure names 109
dynamic binding 57, 163

E

-E compiler option 172
embedded procedure calls 14
-enable pointer_track option 48, 84
error checking 15, 45
ANSI FORTRAN standard 112
error messages 54, 201
ESTIMATED_TRIPS directive 120, 124, 125, 147, 148, 149, 150
-extend_dim option 84

F

-f option 55
FFLAGS macro 109, 110
file-scope static variables 76
flag macros 109
force_object buildfile command 107
FORCE_PARALLEL directive 14
formal parameter
defined 213
formal parameters 25, 51
FORTRAN 66 166
FORTRAN and C
combining 113
FORTRAN libraries 100
FORTRAN source files 99
function
defined 213
function calls
embedded 14
function return values 97
functions
C library 169
further reference xvi

G

global optimization 5
defined 214
global variable
defined 214
global variables 45, 49
propagating 22
-gprof option 90
graph profiling 90

H

hidden alias
defined 214
hidden aliases 12, 15, 49, 51
hints
C 169
FORTRAN 161
Hollerith constants 166

I

-I compiler option 111
-i option 58
IF statements 156
three-way 147
-il 171
-il compiler option 100
incompatible types 45
inconsistent types 97
independence
data 13
informational directives 146
initialization of variables 161
INLINE directive 131, 134, 143
-inline high option 87, 112
-inline low option 87, 112
-inline medium option 87, 112
-inline none option 87, 112
-inline option 86, 139
inline statement 100, 101
INLINE_CALL directive 134, 135, 138
inlining 14, 25, 54, 71, 75, 97, 131, 163
C into FORTRAN 113
defined 214
FORTRAN into C 113
inlining directives 131
instruction cache misses 51

Index

intermediate representation 9
internal operation 6
interprocedural constant propagation 17, 18, 19, 20, 21, 22, 24, 25, 28, 70, 97
interprocedural error messages 54
interprocedural optimization
  defined 214
interprocedural-optimization report 54, 55, 96
invalid aliases 97
invoking apc 53, 55
IPO report 54, 55, 96
-is compiler option 100

L

language scoping rules 5
language-analysis messages 54
lex 11
libraries 100, 106, 117
library functions
  C 169
-library option 56
link optimization 51
-link_sort option 87
linking
  indirectly called procedures 107
loc() function 168
local optimization
  defined 214
loop stride 20

M

macros
  C 170
main procedures
  defined 215
main procedures
  multiple 109
MAIN program
  defined 214
make files 6, 11, 53
make utility 6, 11, 53, 55
malloc 169
math libraries 106
matrix multiply 164
-max_errors option 63
memory-bank conflicts 84
messages
  error and warning 201
  interprocedural error 54
  language-analysis 54
  optimization 54
MFLAGS macro 110
misses, cache 51
mixed language programming 113
mixing C and FORTRAN 113
multilanguage programming 113
multiple declarations 45
multiple directories 110
multiple entries 75
multiple MAIN entries 109

N

-n option 56
name conflicts 75
-no option 57
no_alias statement 100
NO_CLONE directive 84, 144, 145
no_clone statement 100
NO_INLINE directive 94, 133, 135, 145
no_inline statement 100, 101
NO_INLINE_CALL directive 136, 137, 138
NO_RECURRENCE directive 14
non-ANSI aliasing 48
no-recompile option 56
notational conventions xv
note
  on multiple MAIN entires 109
  on source directories 109
-nrep option 55, 64
-nw option 55

O

-o compiler option 172
-o option 56
object files 100, 106
  and psum directives 117
operation
  internal 6
optimization level
  default 2
optimization messages 54
optimization problems 12
optimization report 54, 92, 95
optimization statements 100
optimization strategy 151
option
  - al 167
  -c 159
  -check arrays 166, 167
  -clone all 112
  -clone none 112
  compiler, -B 172
  compiler, -c 100, 172
  compiler, -cfc 106

compiler, -D 172
compiler, -db 106, 153
compiler, -ds 156
compiler, -I 111
compiler, -il 100
compiler, -is 100
compiler, -o 172
compiler, -p 106, 155, 158, 159
compiler, -pa 106, 155, 158, 159
compiler, -pcc 106
compiler, -pg 106
compiler, -S 172
compiler, -vfc 106
-inline 139
-inline high 112
-inline low 112
-inline medium 112
-inline none 112
-show aliases 51
option, -assert subscripts_ok 52
opions
-enable pointer_track 84
options
apc 82
-check 55, 97
-check common 59
-check types 63, 78
-clone 84
-clone all 84
-clone none 84
-compat rff=new 171
compiler 54, 171
-inline 86
-inline high 87
-inline low 87
-inline medium 87
-inline none 87
-n 56
-nrep 55
-nw 55
-or none 55
-path 89
-permit dynamic binding 57
-s 55, 64
-show 55, 65
-show aliases 66
-show all 81
-show arrays 66
-show calls 68
-show clones 69
-show constants 70
-show inline 71
-show pointers 73
-show renames 75
-show types 78
-time 58
-u 82
options statements 109
-or none option 55
ordering documentation xvii
organization of this book xv
out-of-date source files 56
oversubscripting 45, 166

P

-p compiler option 106, 155, 158, 159
-pa compiler option 106, 155, 158, 159
page faults 51
paging 133, 136, 144, 156
parallel optimization 5
defined 215
parallelization 12
parameter
defined 215
parameters
formal 25, 51
-path option 89
-pcc compiler option 106
PDB 89, 159
PDB directory 172
-pdf option 91
performance analyzer 91, 154
-permit dynamic_binding option 57
-permit unresolved_symbols option 57
-pg compiler option 106
pointer classes 97
pointer table 73
pointer tracking 46, 48, 49, 51, 169
defined 215
pointers
Cray-style 168
potential aliases 48
potential recurrence
defined 215
problems of optimization 12
procedural optimization 12, 53
procedure
defined 215
procedure calls
embedded 14
procedure cloning 28, 29, 30, 32, 54, 69, 75, 97, 113
defined 215
directives 139
procedure summaries 107, 117
-prof option 90
profiling 51, 90, 99
program database 5, 9, 12, 89, 159
defined 215
propagation of constants 17, 18, 19, 20, 21, 22, 24, 25, 28, 97
psum directives 120
using 117

psum files 107, 115
psum_args_dealloc directive 123
PSUM_ASGS directive 121
PSUM_KILLS directive 121
PSUM_NAMES directive 124
PSUM_NO_IO directive 125
psum_range_flags directive 123
psum_range_names directive 122
PSUM_REENTRANT directive 124
PSUM_USES directive 122

R

recurrence
  apparent 14
  defined 216
recurrences 12, 49, 51
  apparent 14
reusing clones 32
runtime binding 57

S

-S compiler option 172
-s option 55, 64
scalar arguments 45
scalar directive 156
scalar optimization 5
  defined 216
scoping rules 5
search
  binary 153, 155, 158
short vector lengths 156
-show aliases option 51, 66
-show all option 81
-show arrays option 66
-show calls option 68
-show clones option 69
-show constants option 70
-show inline option 71
-show options 55, 65
-show pointers option 73
-show renames option 75
-show types option 78
silent option 64
simplifying subscripts 156
small trip counts 156
source dependency 89
  defined 216
source directories 108, 109
source file
  defined 216
source files 99, 108
speed of optimization 12
standard libraries 100
statements
  clone 100
  inline 100, 101
  no_alias 100
  no_clone 100
  no_inline 101
  no_inline<255> 100
static binding 163
static variables 49, 75
storage optimization 84
stride
  loop 20
strip mining 18, 19
  defined 216
stub files 115
  compiling 119
  format 117
  simple example file 118
subroutine
  defined 216
subroutine calls
  embedded 14
subscripting problems 166
subscripts
  simplifying 156
supplemental reading xvi
swapping 133, 136, 144
synthesis
  defined 217
synthesis phase 10

T

TAC
  technical assistance center xviii
technical assistance center
  TAC xviii
text files 99
three-way IF statements 147
-time option 58
trip count 156
types
  incompatible 45

U

-u option 82
undersubscripting 45, 166
uninitialized globals 45
uninitialized variables 161
-upto option 88
usage option 82
using apc 55

using psum files 115

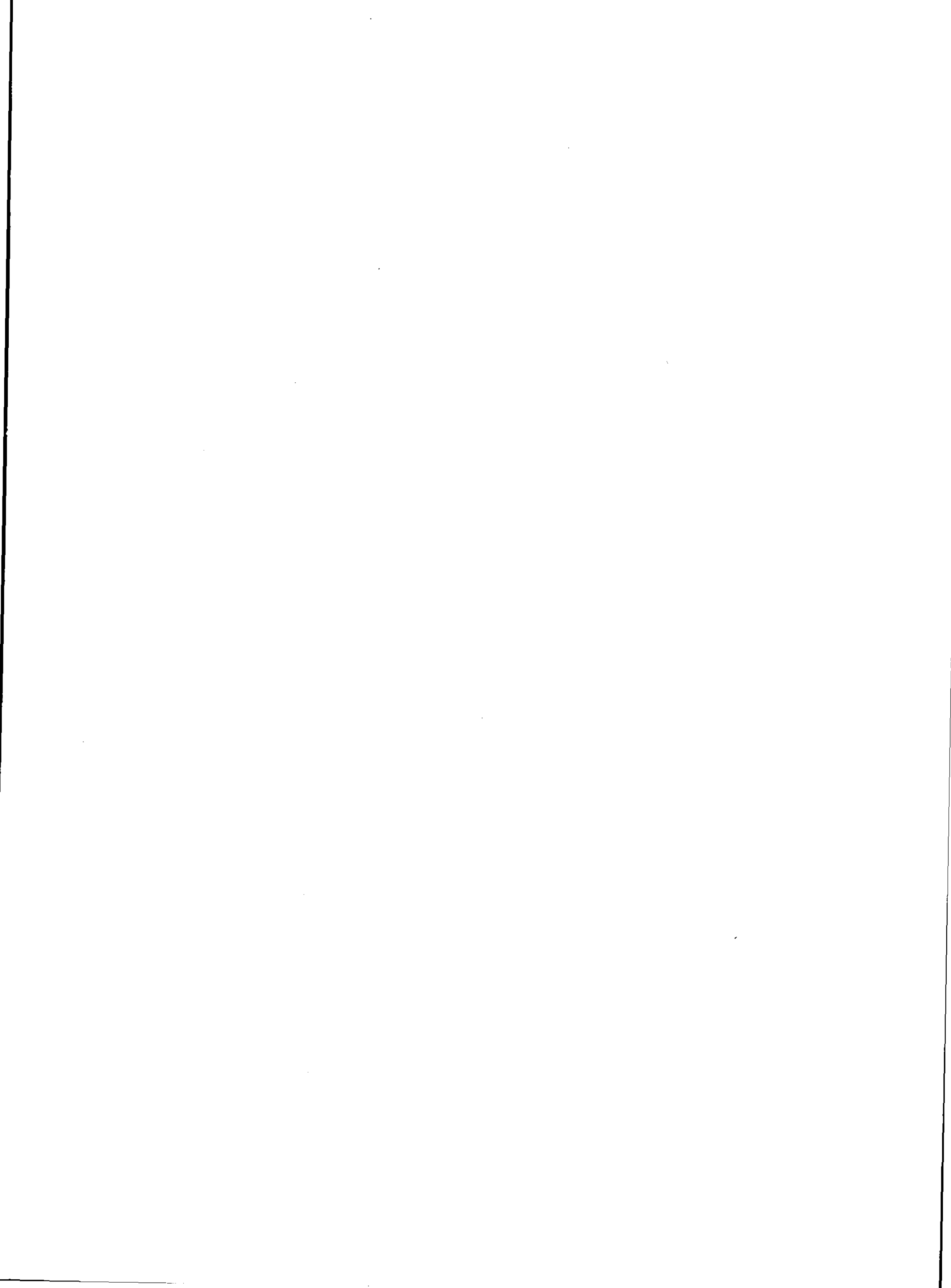
V

-v option 82
var_args buildfile statement 104
variables
CHARACTER 166
VECLIB 106
vector optimization 5
defined 217
vectorization 12, 154
version number 56
-vfc compiler option 106
virtual memory 133, 136, 144, 156
-vn option 56

W

warning messages 201
WHILE statements 147
wrong answers 12, 15

Index



ORDER NUMBER
DSW-401

DOCUMENT NUMBER
720-004030-004

CONVEX PRESS