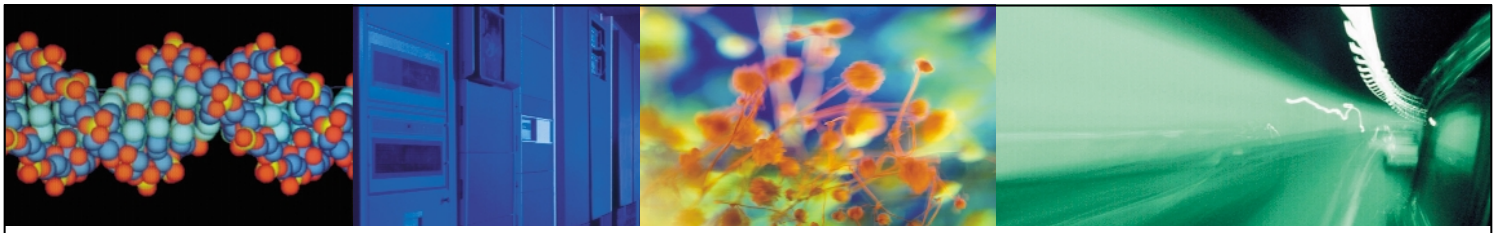

PathScale™

PATHSCALE™ EKOPATH™ COMPILER SUITE

USER GUIDE

VERSION 2.4

Document number: 1-02404-10
Last generated on March 23, 2006

Release version	**New features**
1.4	New Sections 3.3.3.1, 3.7.4, 10.4, 10.5 Added Appendix B: Supported Fortran intrinsics
2.0	New Sections 2.3, 8.9.7, 11.8 Added Chapter 8: Using OpenMP in Fortran New Appendix B: Implementation dependent behavior for OpenMP Fortran Expanded and updated Appendix C: Supported Fortran intrinsics
2.1	Added Chapter 9: Using OpenMP in C/C++, Appendix E: eko man page Expanded and updated Appendix B and Appendix C
2.2	New Sections 3.5.1.4, 3.5.2; combined OpenMP chapters
2.3	Added to reference list in Chapter 1, new Section 8.2 on autoparallelization
2.4	Expanded and updated Section 3.4, Section 3.5, and Section 7.9, Updated Section C.3, added Section C.4 (Fortran intrinstic extensions)

Contents

Chapter 1

Introduction

This User Guide covers how to use the PathScale™ EKOPath™ Compiler Suite compilers; how to configure them, how to use them to optimize your code, and how to get the best performance from them. This guide also covers the language extensions and differences from the other commonly available language compilers.

The PathScale EKOPath Compiler Suite generates both 32-bit and 64-bit code. Generating 64-bit code is the default; to generate 32-bit code use `-m32` on the command line. See the `eko` man page for details.

The information in this guide is organized into these sections:

- Chapter 2 is a quick reference to using the PathScale EKOPath compilers
- Chapter 3 covers the PathScale EKOPath Fortran compiler
- Chapter 4 covers the PathScale EKOPath C/C++ compilers
- Chapter 5 provides suggestions for porting and compatibility
- Chapter 6 is a Tuning Quick Reference, with tips for getting faster code
- Chapter 7 discusses tuning options in more detail
- Chapter 8 covers using autoparallelization and OpenMP in Fortran and C/C++
- Chapter 9 provides an example of optimizing code
- Chapter 10 covers debugging and troubleshooting code
- Appendix A lists environmental variables used with the compilers
- Appendix B discusses implementation dependent behavior for OpenMP Fortran
- Appendix C is a list of the supported Fortran intrinsics
- Appendix D provides a simplified data structure from a Fortran 90 dope vector
- Appendix E is a reference copy of the `eko` man page
- A Glossary of terms associated with the compilers is also included

1.1 Conventions used in this document

These conventions are used throughout the PathScale documentation.

Convention	Meaning
`command`	Fixed-space font is used for literal items such as commands, files, routines, and pathnames.
variable	Italic typeface is used for variable names or concepts being defined.
`user input`	Bold, fixed-space font is used for literal items the user types in. Output is shown in non-bold, fixed-space font.
$	Indicates a command line prompt
#	Command line prompt as root
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.
NOTE:	Indicates important information

1.2 Documentation suite

The PathScale EKOPath Compiler Suite product documentation set includes:

- *The PathScale EKOPath Compiler Suite Install Guide*
- *The PathScale EKOPath Compiler Suite User Guide*
- *The PathScale EKOPath Compiler Suite Support Guide*
- *The PathScale Debugger User Guide*
- *The PathScale Subscription Management User Guide*

There are also online manual pages ("man pages") available describing the flags and options for the PathScale EKOPath Compiler Suite. These man pages are shipped with the Compiler Suite: `eko`, `pathf95`, `pathcc`, `pathCC`. The `pathscale-intro` man page gives an overview of all the various man pages that are included with the Compiler Suite.

Please see the PathScale website at `http://www.pathscale.com/support.html` for further information about current releases and developer support.

In addition, you may want to refer to language reference books for more information on compilers and language usage. Programming and language reference books are often a matter of personal taste. Everyone has a personal preferences in reference books, and this list reflects the variety of opinions found within the PathScale engineering team.

Fortran language:

- *Fortran 95 Handbook: Complete ISO/ANSI Reference* by Jeanne C. Adams, et al., MIT Press, 1997. ISBN 0-262-51096-0
- *Fortran 95 Explained* by Metcalf, M. and Reid, J., Oxford University Press, 1996. ISBN 0-19-851888-8

C language:

- *C Programming Language* by Brian W. Kernighan, Dennis Ritchie, Dennis M. Ritchie, Prentice Hall, 1988, 2nd edition, ISBN 0-13-110362-8
- *C: A Reference Manual* by Samuel P. Harbison, Guy L. Steele, Prentice Hall, 5th Edition, 2002, ISBN 0-130-89592-X
- *C: How to Program* by H.M. Deitel and P.J. Deitel, Prentice Hall, Fourth Edition, 2004 ISBN 0-131-42644-3

C++ Language:

- *The C++ Standard Library A Tutorial and Reference* by Josutis, Nicolai M., 1999. Addison- Wesley, ISBN 0-201-37926-0
- *Effective C++: 55 Specific Ways to Improve Your Programs and Design* by Scott Meyers, Addison-Wesley Professional, 2005, 3rd edition, ISBN 0-321-33487-6
- *More Effective C++: 35 New Ways to Improve Your Programs and Designs* by Scott Meyers, Addison-Wesley Professional, 1995, ISBN 0-201-63371-X
- *Thinking in C++, Volume 1: Introduction to Standard C++* by Bruce Eckel, Prentice Hall, 2nd Edition, 2000, ISBN: 0-139-79809-9 (**NOTE**: There is a later version–2002–available online as a free download.)
- *Thinking in C++, Vol. 2: Practical Programming* by Bruce Eckel, Prentice Hall, Second Edition, 2003, ISBN 0-130-35313-2
- *C++ Inside & Out* by Bruce Eckel, Osborne/McGraw-Hill, 1993, ISBN: 0-07-881809-5
- *C++: How to Program* by H.M. Deitel and P.J. Deitel, Prentice Hall, 2005, 5th edition, ISBN 0-131-85757-6

Other topics:

- *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library* by Scott Meyers, Addison-Wesley Professional, 2001, ISBN 0-201-74962-9

Chapter 2

Compiler Quick Reference

This chapter describes how to get started using the PathScale EKOPath Compiler Suite. The compilers follow the standard conventions of Unix and Linux compilers, produce code that follows the Linux x86_64 ABI, and run on both the AMD64 family of chips and the Intel EM64T family of chips. This means that object files produced by the PathScale EKOPath compilers can link with object files produced by other Linux x86_64-compliant compilers such as Red Hat and SUSE GNU `gcc`, `g++`, and `g77`[1].

We also have full support for the Intel EM64T family of processors. AMD64 is the AMD 64-bit extension to the Intel IA32 architecture, often referred to as "x86". EM64T is the Intel® *Extended Memory 64 Technology* chip family.

2.1 What you installed

The PathScale EKOPath Compiler Suite includes optimizing compilers and runtime support for C, C++, and Fortran.

Depending on the type of subscription you purchased, you enabled some or all of the following:

- PathScale EKOPath C compiler for x86_64 and EM64T architectures
- PathScale EKOPath C++ compiler for x86_64 and EM64T architectures
- PathScale EKOPath Fortran compiler for x86_64 and EM64T architectures
- Documentation
- Libraries
- Subscription Manager client[2]
- Subscription Manager server (optional)[3]

[1] Use of the -ff2c-abi flag may be required for binaries generated by `g77`.
[2] You must have a valid subscription (and associated subscription file) in order to run the compiler.
[3] The PathScale Subscription Manager server is required for floating subscriptions.

- PathScale debugger (`pathdb`)
- GNU binutils

For more details on installing the PathScale EKOPath compilers, see the *PathScale EKOPath Compiler Suite Install Guide*.

2.2 How to invoke the PathScale EKOPath compilers

The PathScale EKOPath Compiler Suite has three different front-ends to handle programs written in C, C++, and Fortran, and it has common optimization and code generation components that interface with all the language front-ends. The language your program uses determines which command (driver) name to use:

Language	Command Name	Compiler Name
C	`pathcc`	PathScale EKOPath C compiler
C++	`pathCC`	PathScale EKOPath C++ compiler
Fortran 77 Fortran 90 Fortran 95	`pathf95`	PathScale EKOPath Fortran compiler

There are online manual pages (“man pages”) with descriptions of the large number of command line options that are available. These man pages are shipped with the Compiler Suite: `eko`, `pathf95`, `pathcc`, `pathCC`. Type `man pathscale_intro` at the command line to see the `pathscale-intro` man page and its overview of the various man pages included with the Compiler Suite.

If invoked with the flag `-v` (or `-version`), the compilers will emit some text that identifies the version. For example:

```
$ pathcc -v
PathScale Compiler Suite(TM): Version 2.4
Built on: 2005-09-29 06:35:06 -0700
gcc version 3.3.1 (PathScale 2.4 driver)
```

You can create a common example program called `world.c`:

```
#include <stdio.h>
main() {
 printf ("Hello World!\n");
}
```

Then you can compile it from your shell prompt very simply:

```
$ pathcc world.c
```

The default output file for the pathcc-generated executable is named `a.out`. You can execute it and see the output:

```
$ ./a.out
Hello World!
```

As with most compilers, you can use the `-o <filename>` option to give your program executable file the desired name.

2.3 Compiling for different platforms

The PathScale EKOPath Compiler Suite currently compiles and optimizes your code for the Opteron processor independent of where the compilation is happening. (This may change in the future.) It will query the machine where the compilation is happening and compile to the best ABI for that machine. These defaults (for the target processor and the ABI) can be overridden by command-line flags or the `compiler.defaults` file.

You can set or change the default platform for compilation using the `compiler.defaults` file, found in `/opt/pathscale/etc` (or `/<install_directory>/pathscale/etc` if you installed in a non-default location). You can use the defaults file to provide a set of additional include or library directories to search, or to specify some default compiler optimization flags.

The compiler refers to the `compiler.defaults` file for options to be used during compilation. If any option conflicts with what is specified on the command line, the compile command line takes precedence. The syntax in `compiler.defaults` file is the same as options on the compiler command line.

The format of the `compiler.defaults` file is simple. Each line can contain compiler options, separated by white space, followed by an optional comment. A comment begins with the # character, and ends at the end of a line. Empty lines and lines containing only comments are skipped.

Options are added to the command line in the order in which they appear in the defaults file. Every option is included unconditionally. For exclusive options, the command line takes precedence over the defaults file. For example, if the defaults file contains the `-O3` option, but the compiler is invoked with `-O2` on the command line, it will behave as if invoked with `-O2` alone, because `-O2` and `-O3` are exclusive options.

For additive options, the command line is used before the defaults file. For example, if the defaults.compiler contains `-I/usr/foo` and the command line contains `-I/usr/bar`, the compiler will behave as if invoked with `-I/usr/bar -I/usr/foo`.

Here is an example defaults file:

```
# EKOPath compiler defaults file.
#
# Set default CPU type to optimize for, since all of our
# systems use the same CPUs.
-march=opteron
# We have a recent Opteron CPU stepping, so it's safe to
# always use SSE3.
```

```
-msse3
# Ensure that the FFTW library is available to users, so
# they don't need to remember where it's installed.
-L/share/fftw3/lib
-I/share/fftw3/include
```

The environment variable PSC_COMPILER_DEFAULTS_PATH, if set, specifies a PATH or a colon-separated list of PATHs, designating where the compiler is to look for the compiler.defaults file. If the environment variable is set, the PATH /opt/pathscale/etc will not be used. If the file cannot be found, then no defaults file will be used, even if one is present in /opt/pathscale/etc.

For more details, see the compiler.defaults man page.

2.3.1 Target options for the 2.4 release

These options, related to ABI, ISA, and processor target, are supported in the 2.4 release:

- -m32
- -m64
- -march= (same as -mcpu= and -mtune=)
- -mcpu= (same as -march= and -mtune=)
- -mtune= (same as -march= and -mcpu=)
- -msse2
- -msse3
- -m3dnow

There are also -mno- versions for these options: -msse2, -msse3, -m3dnow. For example, -mno-msse3. The architectures supported in the 2.4 release are (using the -march= flag in this list):

- -march=(opteron|athlon64|athlon64fx)
- -march=pentium4
- -march=xeon
- -march=em64t

We have also added these two special options. If you want to compile the program so that it can be run on any x86 machine, you can specify anyx86 as the value of the -march, mcpu, or -mtune options.

- -march=anyx86

If you want the compiler to automatically up the target processor based on the machine on which the compilation takes place, you can specify `auto` as the value for the `-march`, `-mcpu`, or `-mtune` options.

- `-march=auto`

Here is a sample `compiler.defaults` file:

```
# Compile for Athlon64 and turn on 3DNow extensions. One
# option per line.
-march=athlon64    # anything after '#' is ignored
-m3dnow
```

These options can also be used on the command line. See the `eko` man page for details.

2.3.2 Defaults flag

This release includes a new flag, `-show-defaults`, which directs the compiler to print out the defaults used related to ABI, ISA, and processor targets. When this flag is specified, the compiler will just print the defaults and quit. No compilation is performed.

```
$ pathcc -show-defaults
```

2.3.3 Compiling for an alternate platform

You will need to compile with the `-march=anyx86` flag if you want to run your compiled executables on both AMD and Intel platforms. (See the `eko` man page for more information about the `-march=` flag.)

To run code generated with the PathScale EKOPath Compiler Suite on a different host machine, you will need to install the runtime libraries on your host machine, or you need to static link your programs when you compile. See Section 2.7 for information on static linking and the *PathScale EKOPath Compiler Suite Install Guide* for information on installing runtime libraries.

2.3.4 Compiling option tool: pathhow-compiled

The PathScale EKOPath Compiler Suite includes a tool that displays the compilation options and compiler version currently being used. The tool is called `pathhow-compiled` and can be found after installation in `/opt/pathscale/bin` (or `/<install_directory>/bin` if you installed to a non-default location).

When a `.o` file, archive, or an executable is passed to `pathhow-compiled`, it will display the compilation options for each `.o` file constituting the argument file. This includes any linked archives.

For example, if you compile the file `myfile.c` with `pathcc` (producing a file `myfile.o`) and then use the `pathhow-compiled` tool:

```
$ pathhow-compiled myfile.o
```

The output would look something like this:

```
EKOPath Compiler Version 2.4 compiled myfile.c with options:
   -O2 -march=opteron -msse2 -mno-sse3 -mno-3dnow -m64
```

2.4 Input file types

The name for a source file usually has the form `filename.ext`, where `ext` is a one to three character extension used on a source code file that can have various meanings:

Extension	Implication to the driver
`.c`	C source file that will be preprocessed

`.C`
`.cc`
`.cpp`
`.cxx`
	C++ source file that will be preprocessed

`.f`
`.f90`
`.f95`
	Fortran source file
	`.f` is fixed format, no preprocessor
	`.f90` is freeform format, no preprocessor
	`.f95` is freeform format, no preprocessor

`.F`
`.F90`
`.F95`
	Fortran source file
	`.F` is fixed format, invokes preprocessor
	`.F90` is freeform format, invokes preprocessor
	`.F95` is freeform format, invokes preprocessor

For Fortran files with the extensions `.f`, `.f90`, or `.f95` you can use `-ftpp` (to invoke the Fortran preprocessor) or `-cpp` (to invoke the C preprocessor) on the `pathf95` command line. The default preprocessor for files with `.F`, `.F90`, or `.F95` extensions, is `-cpp`. See Section 3.4.1 for more information on preprocessing.

The compiler drivers can use the extension to determine which language front-end to invoke. For example, some mixed language programs can be compiled with a single command:

```
# pathf95 stream_d.f second_wall.c -o stream
```

The `pathf95` driver will use the `.c` extension to know that it should automatically invoke the C front-end on the `second_wall.c` module and link the generated object files into the `stream` executable.

NOTE: GNU make does not contain a rule for generating object files from Fortran `.f90` files. You can add the following rules to your project Makefiles to achieve this:

```
$.o:   %.f90
       $(FC) $(FFLAGS) -c $<
$.o:   %.F90
       $(FC) $(FFLAGS) -c $<
```

You may need to modify this for your project, but in general the rules should follow this form.

For more information on compatibility and porting existing code, see Section 5. Information on GCC compatibility and a wrapper script that you can use for your build packages can be found in Section 5.6.1.

2.5 Other input files

Other possible input files, common to both C/C++ and Fortran, are assembly-language files, object files, and libraries as inputs on the command line.

Extension	Implication to the driver
`.i`	preprocessed C source file
`.ii`	preprocessed C++ source file
`.s`	assembly language file
`.o`	object file
`.a`	a static library of object files
`.so`	a library of shared (dynamic) object files

2.6 Common compiler options

The PathScale EKOPath Compiler Suite has command line options that are similar to many other Linux or Unix compilers:

Option	What it does
`-c`	Generates an intermediate object file for each source file, but doesn't link
`-g`	Produces debugging information to allow full symbolic debugging
`-I<dir>`	Adds <*path*> to the directories searched by preprocessor for include file resolution.
`-l<library>`	Searches the library specified during the linking phase for unresolved symbols
`-L<dir>`	Adds <path> to the directories searched during the linking phase for libraries
`-lm`	Links using the `libm` math library. This is typically required in C programs that use functions such as `exp()`, `log()`, `sin()`, `cos()`.
`-o <filename>`	Generates the named executable (binary) file
`-O3`	Generates a highly optimized executable, generally numerically safe
`-O or -O2`	Generates an optimized executable that is numerically safe. (This is also the default if no `-O` flag is used.)
`-pg`	Generates profile information suitable for the analysis program `pathprof`

Many more options are available and described in the man pages (`pathscale_intro, pathcc`, `pathf95`, `pathCC`, `eko`) and Chapter 7 in this document.

2.7 Shared libraries

The PathScale EKOPath Compiler Suite includes shared versions of the runtime libraries that the compilers use. The shared libraries are packaged in the `pathscale-compilers-libs` package. The compiler will use these shared libraries by default when linking executables and shared objects. As a result, if you link a program with these shared libraries, you must install them on systems where that program will run.

You should continue to use the static versions of the runtime libraries if you wish to obtain maximum portability or peak performance. The latter is the case because the compiler cannot optimize shared libraries as aggressively as static libraries. Shared libraries are compiled using position-independent code, which limits some opportunities for optimization, while our static libraries are not compiled this way.

To link with static libraries instead of shared libraries, use the `-static` option. For example the following code is linked using the shared libraries.

```
$ pathcc -o hello hello.c
$ ldd hello
       libpscrt.so.1 => /opt/pathscale/lib/2.3.99/libpscrt.so.1
(0x0000002a9566d000)
       libmpath.so.1 => /opt/pathscale/lib/2.3.99/libmpath.so.1
(0x0000002a9576e000)
       libc.so.6 => /lib64/libc.so.6
(0x0000002a9588b000)
       libm.so.6 => /lib64/libm.so.6
(
0x0000002a95acd000)
       /lib64/ld-linux-x86-64.so.2 => /lib64/ld-linux-x86-64.so.2
(0x0000002a95556000)
$
```

If you use the `-static` option, notice that the shared libraries are no longer required.

```
$ pathcc -o hello hello.c -static
$ ldd hello
       not a dynamic executable
$
```

2.8 Large file support

The Fortran runtime libraries are compiled with large file support. PathScale does not provide any runtime libraries for C or C++ that do I/O, so large file support is provided by the libraries in the Linux distribution being used.

2.9 Large object support

The PathScale compilers currently support two memory models: small and medium.

The default memory model on x86_64 systems, and the default for the compilers, is small (equivalent to GCC's `-mcmodel=small`). This means that offsets of code and data within binaries are represented as signed 32-bit quantities. In this model, all code in an executable must total less than 2GB, and all the data must also be less than 2GB. Note that by data, we mean the static and unlimited static data (BSS) that are compiled into an executable, not data allocated dynamically on the stack or from the heap.

Pointers are 64-bits however, so dynamically allocated memory may exceed 2GB. Programs can be statically or dynamically linked.

Additionally the compilers support the medium memory model with the use of the option `-mcmodel=medium` on all of the compilation and link commands. This means that offsets of code within binaries are represented as signed 32-bit quantities. The offsets for data within the binaries are represented as signed

64-bit quantities. In this model, all code in an executable must come to less than 2GB in total size. The data, both static and BSS, are allowed to exceed 2GB in size.

As with the small memory model, pointers are also signed 64-bit quantities and may exceed 2 GB in size.

NOTE: The PathScale EKOPath compilers do not support the use of the `-fPIC` option flag in combination with the `-mcmodel=medium` option. The code model `medium` is not supported in `PIC` mode.

The PathScale compilers support `-mcmodel=medium` and `-fPIC` in the same way that GCC does. When building shared libraries, only `-fPIC` should be used. The option `-mcmodel=medium` but not `-fPIC` when compiling and linking the main program.

The reasoning behind this is that because the shared library is self-contained, it does not know about the fixed addresses of the data in the program that it is linked with. The library will only access the program data through pointers, and such pointer data accesses are not affected by the value of the `mcmodel` option. The `mcmodel` value only affects the addressing of data with fixed addresses. When these addresses are larger than 2GB, the compiler has to generate longer sequences of instructions. Thus, it does not want to do that unless the `-mcmodel=medium` flag is given.

See 10.4 for more information on using large objects, and your GCC 3.3.1 documentation for more information on this topic.

2.9.1 Support for "large" memory model

At this time the PathScale compilers do not support the large memory model. The significance is that the code offsets must fit within the signed 32-bit address space. To determine if you are close to this limit, use the Linux `size` command.

```
$ size bench
   text    data     bss     dec     hex filename
 910219    1448    3192   914859   df5ab bench
```

If the total value of the `text` segment is close to 2GB, then the size of the memory model may be an issue for you. We believe that codes that are this large are extremely rare and would like to know if you are using such an application.

The size of the `bss` and `data` segments are addressed by using the medium memory model.

2.10 Debugging

The flag `-g` tells the PathScale EKOPath compilers to produce data in the form used by modern debuggers, such as GDB and PathScale's debugger, `pathdb`. This format is known as DWARF 2.0 and is incorporated directly into the object files.

Code that has been compiled using `-g` will be capable of being debugged using `pathdb`, GDB, or other debuggers.

The `-g` option automatically sets the optimization level to `-O0` unless an explicit optimization level is provided on the command line. Debugging of higher levels of optimization is possible, but the code transformation performed by the optimizations may make it more difficult.

See the individual chapters on the PathScale EKOPath Fortran and C /C++ compilers for more language-specific debugging information, and Section 10 for debugging and troubleshooting tips. See the *PathScale Debugger User Guide* for more information on `pathdb`.

2.11 Profiling: Locate your program's hot spots

To figure out where and how to tune your code, use the `time` tool to get a rough estimate and determine if the issue is system load, application load, or a system resource that is slowing down your program, and then use `pathprof` to find the program's hot spots.

NOTE: The `pathprof` and `pathcov` programs included with the compilers are symbolic links to your system's `gcov` and `gprof` executibles. Also note that the `pathprof` tool will generate a segmentation fault when used with OpenMP applications that are run with more than one thread. There is no current workaround for `pathprof` (or `gprof`).

The `time` tool provides the elapsed (or `wall`) time, user time, and system time of your program. Its usage is typically: `time ./<program args>`. Elapsed time is usually the measurement of interest, especially for parallel programs, but if your system is busy with other loads, then user time might be a more accurate estimate of performance than elapsed time. If there is substantial system time being used and you don't expect to be using substantial non-compute resources of the system, you should use a kernel profiling tool to see what is causing it.

Often a program has "hot spots," a few routines or loops that are responsible for most of the execution time. Profilers are a common tool for finding these hot spots in a program. Once you find the hot spots in your program, you can improve your code for better performance, or use the information to help choose which compiler flags are likely to lead to better performance.

The PathScale EKOPath Compiler Suite includes a symbolic link to the standard Linux profiler `gprof` (`pathprof`). There are more details and an example using `pathprof` later in Chapter 9, but the following steps are all that are needed to get started in profiling:

1. Add the `-pg` flag to both the compile and link steps with the PathScale EKOPath compilers. This generates an instrumented binary.
2. Run the program executable with the input data of interest. This creates a `gmon.out` file with the profile data.
3. Run `pathprof <program-name>` to generate the profiles. The standard output of `pathprof` includes two tables:

(a) a flat profile with the time consumed in each routine and the number of times it was called, and

(b) a call-graph profile that shows, for each routine, which routines it called and which other routines called it. There is also an estimate of the inclusive time spent in a routine and all of the routines called by that routine.

See Chapter 9 for a more detailed example of profiling.

2.12 Taskset: Assigning a process to a specific CPU

To improve the performance of your application on multiprocessor machines, it is useful to assign the process to a specific CPU. The tool used to do this is `taskset`, which can be used to retrieve or set a process' affinity. This command is part of the `schedutils` package/RPM and may or may not be installed as part of your default configuration.

The CPU affinity is represented as a bitmask, typically given in hexadecimal. Assigning a process to a specific CPU prevents the Linux scheduler from moving or splitting the process.

Example:

```
$ taskset 0x00000001
```

This would assign the process to processor #0.

If an invalid mask is given, an error is returned, so when `taskset` returns, it is guaranteed that the program has been scheduled on a valid and legal CPU. See the `taskset(1)` man page for more information.

NOTE: Some of the Linux distributions supported by the PathScale compilers do not contain the `schedutils` package/RPM.

Chapter 3

The PathScale EKOPath Fortran compiler

The PathScale EKOPath Fortran compiler supports Fortran 77, Fortran 90, and Fortran 95. The PathScale EKOPath Fortran compiler:

- Conforms to ISO/IEC 1539:1991 Programming languages–Fortran (Fortran 90)
- Conforms to the more recent ISO/IEC 1539-1:1997 Programming languages–Fortran (Fortran 95)
- Supports legacy FORTRAN 77 (ANSI X3.9-1978) programs
- Provides support for common extensions to the above language definitions
- Links binaries generated with the GNU Fortran 77 compiler
- Generates code that complies with the x86_64 ABI and the 32-bit x86 ABI

3.1 Using the Fortran compiler

To invoke the PathScale EKOPath Fortran compiler, use this command:

```
$ pathf95
```

By default, the compiler will treat input files with an `.F` suffix or `.f` suffix as fixed-form files. Files with an `.F90`, `.f90`, `.F95`, or `.f95` suffix are treated as free-form files. This behavior can be overridden using the `-fixedform` and `-freeform` switches. See Section 3.1.1 for more information on fixed-form and free-form files.

By default, all files ending in `.F`, `.F90`, or `.F95` are first preprocessed using the C preprocessor (`-cpp`). If you specify the `-ftpp` option, all files are preprocessed using the Fortran preprocessor (`-ftpp`), regardless of suffix. See Section 3.4.1 for more information on preprocessing.

Invoking the compiler without any options instructs the compiler to use optimization level `-O2`. These three commands are equivalent:

```
$ pathf95 test.f90
$ pathf95 -O test.f90
$ pathf95 -O2 test.f90
```

Using optimization level `-O0` instructs the compiler to do no optimization. Optimization level `-O1` performs only local optimization. Level `-O2`, the default, performs extensive optimizations that will always shorten execution time, but may cause compile time to be lengthened. Level `-O3` performs aggressive optimization that may or may not improve execution time. See Section 7.1 for more information about the `-O` flag.

Use the `-ipa` switch to enable inter-procedural analysis:

```
$ pathf95 -c -ipa matrix.f90
$ pathf95 -c -ipa prog.f90
$ pathf95 -ipa matrix.o prog.o -o prog
```

Note that the link line also specifies the `-ipa` option. This is required to perform the IPA link properly.

See Section 7.3 for more information on IPA.

NOTE: The compiler typically allocates data for Fortran programs on the stack for best performance. Some major Linux distributions impose a relatively low limit on the amount of stack space a program can use. When you attempt to run a Fortran program that uses a large amount of data on such a system, it will print an informative error message and abort. You can use your shell's "`ulimit`" (`bash`) or "`limit`" (`tcsh`) command to increase the stack size limit to a point where the program no longer crashes, or remove the limit entirely. See Section 3.11 for more information on Fortran compiler stack size.

3.1.1 Fixed-form and free-form files

Fixed-form files follow the obsolete Fortran standard of assigning special meaning to the first 6 character positions of each line in a source file.

If a `C`, `!` or `*` character is present in the first character position on a line, that specifies that the remainder of the line is to be treated as a comment. If a `!` is present at any character position on a line except for the 6th character position, then the remainder of that line is treated as a comment. Lines containing only blank characters or empty lines are also treated as comments.

If any character other than a blank character is present in the 6th character position on a line, that specifies that the line is a continuation from the previous line. The Fortran standard specifies that no more than 19 continuation lines can follow a line, but the PathScale compiler supports up to 499 continuation lines.

Source code appears between the 7th character position and the 72nd character position in the line, inclusive. Semicolons are used to separate multiple

statements on a line. A semicolon cannot be the first non-blank character between the 7th character position and the 72nd character position.

Character positions 1 through 5 are for statement labels. Since statement labels cannot appear on continuation lines, the first five entries of a continuation line must be blank.

Free-form files have fewer limitations on line layout. Lines can be arbitrarily long, and continuation is indicated by placing an ampersand (&) at the end of the line before the continuation line. Statement labels can be placed at any character position in a line, as long as it is preceded by blank characters only. Comments start with a `!` character anywhere on the line.

3.2 Modules

When a Fortran module is compiled, information about the module is placed into a file called `MODULENAME.mod`. The default location for this file is in the directory where the command is executed. This location can be changed using `-module` option. The `MODULENAME.mod` file allows other Fortran files to use procedures, functions, variables, and any other entities defined in the module. Module files can be considered similar to C header files.

Like C header files, you can use the `-I` option to point to the location of module files:

```
$ pathf95 -I/work/project/include -c foo.f90
```

This instructs the compiler to look for `.mod` files in the `/work/project/include` directory. If `foo.f90` contains a '`use arith`' statement, the following locations would be searched:

```
/work/project/include/ARITH.mod
./ARITH.mod
```

3.3 Extensions

The PathScale EKOPath Fortran compiler supports a number of extensions to the Fortran standard, which are described in this section.

3.3.1 Promotion of REAL and INTEGER types

Section 5 has more information about porting code, but it is useful to mention the following option that you can use to help in porting your Fortran code.

`-r8 -i8` Respectively promotes the default representation for REAL and INTEGER type from 4 bytes to 8 bytes. Useful for porting from Cray code when integer and floating point data is 8 bytes long by default. Watch out for type mismatches with external libraries.

NOTE: The `-r8` and `-i8` flags only affect default reals and integers, not variable declarations or constants that specify an explicit `KIND`. This can cause incorrect results if a 4-byte default real or integer is passed into a subprogram that declares a `KIND=4` integer or real. Using an explicit `KIND` value like this is unportable and is not recommended. Correct usage of `KIND` (i.e. `KIND=KIND(1)` or `KIND=KIND(0.0d0)`) will not result in any problems.

3.3.2 Cray pointers

The Cray pointer is a data type extension to Fortran to specify dynamic objects, different from the Fortran pointer. Both Cray and Fortran pointers use the `POINTER` keyword, but they are specified in such a way that the compiler can differentiate between them.

The declaration of a Cray pointer is:

```
POINTER ( <pointer>, <pointee> )
```

Fortran pointers are declared using:

```
POINTER :: [ <object_name> ]
```

PathScale's implementation of Cray Pointers is the Cray implementation, which is a stricter implementation than in other compilers. In particular, the PathScale EKOPath Fortran compiler does not treat pointers exactly like integers. The compiler will report an error if you do something like `p = ((p+7)/8)*8` to align a pointer.

3.3.3 Directives

Directives within a program unit apply only to that program unit, reverting to the default values at the end of the program unit. Directives that occur outside of a program unit alter the default value, and therefore apply to the rest of the file from that point on, until overridden by a subsequent directive.

Directives within a file override the command line options by default. To have the command line options override directives, use the command line option:

```
-LNO:ignore_pragmas
```

For the 2.4 release, the PathScale EKOPath Compiler Suite supports the following prefetch directives.

3.3.3.1 F77 or F90 prefetch directives

C*$* PREFETCH (N[,N]) Specify prefetching for each level of the cache. The scope is the entire function containing the directive. `N` can be one of the following values:

0 Prefetching off (the default)

1 Prefetching on, but conservative

2 Prefetching on, and aggressive (the default when prefetch is on)

C*$* PREFETCH_MANUAL (N) Specify if manual prefetches (through directives) should be respected or ignored. Scope: Entire function containing the directive. `N` can be one of the following values:

0 Ignore manual prefetches

1 Respect manual prefetches

C*$* PREFETCH_REF_DISABLE=A [, size=num] This directive explicitly disables prefetching all references to array `A` in the current function. The auto-prefetcher runs (if enabled) ignoring array `A`. The size is used for volume analysis. Scope: Entire function containing the directive.

size=num is the size of the array references in this loop, in Kbyte. This is an optional argument and must be a constant.

C*$* PREFETCH_REF=array-ref,[stride=[str] [,str]], [level=[lev] [,lev]], [kind=[rd/wr]], [size=[sz]] This directive generates a single prefetch instruction to the specified memory location. It searches for array references that match the supplied reference in the current loop-nest. If such a reference is found, that reference is connected to this prefetch node with the specified parameters. If no such reference is found, this prefetch node stays free-floating and is scheduled "loosely".

All references to this array in this loop-nest are ignored by the automatic prefetcher (if enabled).

If the size is supplied, then the auto-prefetcher (if enabled) reduces the effective cache size by that amount in its calculations.

The compiler tries to issue one prefetch per stride iteration, but cannot guarantee it. Redundant prefetches are preferred to transformations (such as inserting conditionals) which incur other overhead.

Scope: No scope. Just generates a prefetch instruction.

The following arguments are used with this option:

array-ref Required. The reference itself, for example, A(i, j).

str Optional. Prefetch every `str` iterations of this loop. The default is `1`.

lev Optional. The level in memory hierarchy to prefetch. The default is `2`. If lev=1, prefetch from L2 to L1 cache.If lev=2, prefetch from memory to L1 cache.

rd/wr Optional. The default is read/write.

sz Optional. The size (in Kbytes) of the array referenced in this loop. This must be a constant.

3.3.3.2 Changing optimization using directives

Optimization flags can now be changed via directives in the user program.

In Fortran, the directive is used in the form:

```
C*$* options <"list-of-options">
```

Any number of these can be specified inside function scopes. Each affects only the optimization of the entire function in which it is specified. The literal string can also contain an unlimited number of different options separated by spaces and must include the enclosing quotes. The compilation of the next function reverts back to the settings specified in the compiler command line.

In this release, there are limitations to the options that are processed in this options directive, and their effects on the optimization.

- There is no warning or error given for options that are not processed.
- These directives are processed only in the optimizing backend. Thus, only options that affect optimizations are processed.
- In addition, it will not affect the phase invocation of the backend components. For example, specifying `-O0` will not suppress the invocation of the global optimizer, though the invoked backend phases will honor the specified optimization level.
- Apart from the optimization level flags, only flags belonging to the following option groups are processed: `-LNO`, `-OPT` and `-WOPT`.

3.4 Compiler and runtime features

The EKOPath compiler offers three different preprocessing options; `-cpp`, `-ftpp`, and now `-fcoco`.

3.4.1 Preprocessing source files with `-cpp`

Before being passed to the compiler front-end, source files are optionally passed through a source code preprocessor. The preprocessor searches for certain directives in the file and, based on these directives, can include or exclude parts of the source code, include other files or define and expand macros. By default, Fortran `.F`, `.F90`, and `.F95` files are passed through the C preprocessor `-cpp`.

3.4.2 Preprocessing source files with `-ftpp`

The Fortran preprocessor `-ftpp` accepts many of the same "#" directives as the C preprocessor but differs in significant details (for example, it does not allow C-style comments beginning with "`/*`" to extend across multiple lines.) You should use the `-cpp` option if you wish to use the C preprocessor on Fortran source files ending in `.f`, `.f90`, or `.f95`. These files will not be preprocessed unless you use either `-ftpp` (to select the Fortran preprocessor) or `-cpp` (to select the C preprocessor) on the command line.

3.4.3 Preprocessing source files with `-fcoco`

Beginning with release 2.4, the EKOPath Fortran compiler now supports the ISO/IEC 1539-3 conditional compilation preprocessor. When you use the `-fcoco` option, the compiler runs this preprocessor on each individual source file before compiling that source file, overriding the default whereby files suffixed with `.F`, `.F90`, or `.F95` are preprocessed with `cpp` but files suffixed with `.f`, `.f90`, or `.f95` are not preprocessed.

The ISO/IEC standard does not specify any command-line options for the preprocessor, but as an extension, we pass `-I` and `-D` options to it, just as we do for the `-cpp` and `-ftpp` preprocessors. As with the other preprocessors, an option like `-Isubdir` (no trailing "`/`" is needed) tells the preprocessor to add `subdir` to the list of directories in which it will search for included files.

Unlike the `-cpp` and `-ftpp` preprocessors, this one requires that its identifiers be declared with a data type, so an option like `-DIVAR=5` declares a constant (not a variable) `IVAR` with the type `integer` and the value `5`, while an option like `-DLVAR` declares a constant `LVAR` with the type `logical` and the value "`.true.`". Only integer and logical constants are allowed. You can use the `-D` option to override the value of a constant declaration for that identifier which might appear in the source file.

The standard requires that the preprocessor read a "`setfile`" capable of defining constants, variables and modes of operation, but it does not specify how to find the `setfile`. If you use `-fcoco`, the preprocessor looks for `coco.set` in the current directory. If no such file exists, the preprocessor quietly proceeds without it. If you use an option like `-fcoco=somedir/mysettings`, the preprocessor looks for file `somedir/mysettings`. You cannot use the `-D` option to override a constant declaration which appears in the setfile.

The open-source package on which this feature is based does provide additional extensions and command-line options, described at `http://users.erols.com/dnagle/coco.html`. To pass those options through the compiler driver to the preprocessor, you can use the `-Wp,`*<options>* flag. For example, you can use `-Wp,-m` to pass the `-m` option to the preprocessor to turn off macro preprocessing. Note that the instructions given in that web page for passing file names to the preprocessor and identifying the `setfile` are not relevant when you use the EKOPath compiler, since the compiler automatically passes each source file name to the preprocessor for you, captures the preprocessor output for compilation, and identifies the `setfile` as described in the preceding paragraphs.

More information about the `-fcoco` option can be found in the `eko` man page.

3.4.3.1 Pre-defined macros

The PathScale compiler pre-defines some macros for preprocessing code. When you use the C preprocessor `cpp` with Fortran, or rely on the `.F`, `.F90`, and `.F95` suffixes to use the default `cpp` preprocessor, the PathScale compiler uses the same preprocessor it uses for C, with the addition of the following macros:

```
LANGUAGE_FORTRAN
_LANGUAGE_FORTRAN 1
_LANGUAGE_FORTRAN90 1
LANGUAGE_FORTRAN90 1
__unix 1
unix 1
__unix__ 1
```

NOTE: When using an optimization level at `-O1` or higher, the compiler will set and use the `__OPTIMIZE__` macro with `cpp`.

See the complete list of macros for `cpp` in Section 4.2.1.1.

If you use the Fortran preprocessor `-ftpp`, *only* these five macros are defined for you:

```
LANGUAGE_FORTRAN 1
__LANGUAGE_FORTRAN90 1
LANGUAGE_FORTRAN90 1
__unix 1
unix 1
```

NOTE: By default, Fortran uses `cpp`. You must specify the `-ftpp` command-line switch with Fortran code to use the Fortran preprocessor.

This command will print to `stdout` all of the "`#define`"s used with `-cpp` on a Fortran file:

```
$ echo > junk.F90; pathf95 -cpp -Wp,-dD -E junk.F90
```

There is no corresponding way to find out what is defined by the default Fortran preprocessor (`-ftpp`). See Section 3.4.3.1 for information on how to find pre-defined macros in C and C++.

No macros are predefined for the `-fcoco` preprocessor.

3.4.4 Error numbers: the Explain command

The `explain` program is a compiler and runtime error message utility that prints a more detailed message for the numerical compiler messages you may see.

When the Fortran compiler or runtime prints out an error message, it prefixes the `message` with a string in the format "subsystem-number". For example, "`pathf95-0724`". The "`pathf95-0724`" is the message ID string that you will give to `explain`.

When you type `explain pathf95-0724`, the `explain` program provides a more detailed error message:

```
$ explain pathf95-0724
Error : Unknown statement. Expected assignment statement
but found "%s" instead of "=" or "=>".

The compiler expected an assignment statement
but could not find an assignment or pointer assignment
operator at the correct point.
```

Another example:

```
$ explain pathf95-0700
Error : The intrinsic call "%s" is being made with illegal
arguments.

A function or subroutine call which invokes the name of
an intrinsic procedure does not match any specific intrinsic.
All dummy arguments without the OPTIONAL attribute must
match in type and rank exactly.
```

The `explain` command can also be used with `iostat=` error numbers. When the `iostat=` specifier in a Fortran I/O statement provides an error number such as `4198`, or when the program prints out such an error number during execution, you can look up its meaning using the `explain` command by prefixing the number with `lib-`, as in `explain lib-4198`.

For example:

```
$ explain lib-4098
A BACKSPACE is invalid on a piped file.
A Fortran BACKSPACE statement was attempted on a named or unnamed
pipe (FIFO file) that does not support backspace.
Either remove the BACKSPACE statement or change the file so that it
is not a pipe.
See the man pages for pipe(2), read(2), and write(2).
```

3.4.5 Fortran 90 dope vector

Modern Fortran provides constructs that permit the program to obtain information about the characteristics of dynamically allocated objects such as the size of arrays and character strings. Examples of the language constructs that return this information include the `ubound` and the `size` intrinsics.

To implement these constructs, the compiler may maintain information about the object in a data structure called a *dope vector*. If there is a need to understand this data structure in detail, it can be found in the source distribution in the file `clibinc/cray/dopevec.h`. See Appendix D for an example of a simplified version of that data structure, extracted from that file.

3.4.6 Bounds checking

The PathScale EKOPath Fortran compiler can perform bounds checking on arrays. To enable this feature, use the `-C` option:

```
$ pathf95 -C gasdyn.f90 -o gasdyn
```

The generated code checks all array accesses to ensure that they fall within the bounds of the array. If an access falls outside the bounds of the array, you will get a warning from the program printed on the standard error at runtime:

```
$ ./gasdyn
lib-4961 : WARNING
  Subscript 20 is out of range for dimension 1 for array
  'X' at line 11 in file 't.f90' with bounds 1:10.
```

If you set the environment variable `F90_BOUNDS_CHECK_ABORT` to `YES`, then the resulting program will abort on the first bounds check violation.

Obviously, array bounds checking will have an impact on code performance, so it should be enabled only for debugging and disabled in production code that is performance sensitive.

3.4.7 Pseudo-random numbers

The pseudo-random number generator (PRNG) implemented in the standard PathScale EKOPath Fortran library is a non-linear additive feedback PRNG with a 32-entry long seed table. The period of the PRNG is approximately `16*((2**32)-1)`.

3.5 Mixed code

If you have a large application that mixes Fortran code with code written in other languages, and the main entry point to your application is from C or C++, you can optionally use `pathcc` or `pathCC` to link the application, instead of `pathf95`. If you do, you must manually add the Fortran runtime libraries to the link line.

As an example, you might do something like this:

```
$ pathCC -o my_big_app file1.o file2.o -lpathfortran
```

3.5.1 Calls between C and Fortran

In calls between C and Fortran, the two issues are:

- Mapping Fortran procedure names onto C function names and
- Matching argument types

Normally a `pathf90` procedure name "`x`" not containing an underscore creates a linker symbol "`x_`", and a `pathf90` name "`x_y`" containing an underscore creates a linker symbol "`x_y__`" (note the second underscore). A `pathcc` function name, by contrast, does not append any underscores when creating a linker symbol.

You can write your C code to conform to this: use "`x_`" in C so that it will match Fortran's "`x`". Or you can use the `-fdecorate` option, described in `man pathf90`, to provide a mapping from each Fortran name onto some (possibly quite different) linker symbol. Or you can use the `-fno-underscoring` option, but in many cases that will create symbols that conflict with those in the Fortran and C runtime libraries, so it is not the preferred choice.

Normally `pathf90` passes arguments by reference, so C needs to use pointers in order to interoperate with Fortran. In many cases you can use the `%val()` intrinsic function in Fortran to pass an argument by value.

The programmer must be careful to match argument data types. For instance, `pathf90 integer*4` matches C `int`, `integer*8` matches C `long long`, `real` matches C `float` (provided the C function has an explicit prototype) and `doubleprecision` matches C `double`. Fortran `character` is problematic because in addition to passing a pointer to the first character, it appends an integer length-count argument to the end of the usual argument list. Fortran `Cray` pointers, declared with the `pointer` statement, correspond to C pointers, but Fortran 90 pointers, declared with the `pointer` attribute, are unique to Fortran.

The `sequence` keyword makes it more likely that a Fortran 90 structure will use the same layout as a C structure, although it is wise to verify this by experiment in each case. For arrays, it is wise to limit the interface to the kinds of arrays provided in Fortran 77, since the arrays introduced in Fortran 90 add to the data structures information that C cannot understand.

Thus, for example, an argument "`a(5,6)`" or "`a(n)`" or "`a(1:*)`" (where "`n`" is a dummy argument) will pass a simple pointer that corresponds well to a C array, whereas "`a(:,:)`" or an allocatable array or a Fortran 90 pointer array does not correspond to anything in C.

NOTE: Fortran arrays are placed in memory in column-major order whereas C arrays use row-major order. And, of course, one must adjust for the fact that C array indices originate a zero, whereas Fortran array indices originate at 1 by default but can be declared with other origins instead.

Calls between C++ and Fortran are more difficult, for the same reason that calls between C and C++ are difficult: the C++ compiler must "mangle" symbol names to implement overloading, and the C++ compiler must add to data structures various information (such as virtual table pointers) that other languages cannot understand. The simplest solution is to use the `extern "C"` declaration within the C++ source code to tell it to generate a C-compatible interface, which reduces the problem to that of interfacing C and Fortran.

3.5.1.1 Example: Calls between C and Fortran

Here are three files your can compile and execute, that demonstrate calls between C and Fortran.

This is the C source code (`c_part.c`):

```
#include <stdio.h>
#include <alloca.h>
#include <string.h>
extern void f1_(char *c, int *i, long long *ll, float *f, double *d,
  int *l, int c_len);
/* Demonstrate how to call Fortran from C */
void call_fortran() {
  char *c = "hello from call_fortran";
  int i = 123;
  long long ll = 456ll;
  float f = 7.8;
double d = 9.1;
int nonzero = 10; /* Any nonzero integer is .true. in Fortran */
f1_(c, &i, &ll, &f, &d, &nonzero, strlen(c));
}
/* C function designed to be called from Fortran, passing
arguments by * reference */
void c_reference__(double *d1, float *f1, int *i1, long long *i2, char
*c1,
  int *l1, int *l2, char *c2, char *c3, int c1_len, int c2_len, int
c3_len) {
/* A fortran string has no null terminator, so make a local copy
and add
* a terminator. Depending on the situation, it might be preferable to
* put the terminator in place of the first trailing blank. */
char *null_terminated_c1 = memcpy(alloca(c1_len + 1), c1, c1_len);
char *null_terminated_c2 = memcpy(alloca(c2_len + 1), c2, c2_len);
char *null_terminated_c3 = memcpy(alloca(c3_len + 1), c3, c3_len);
null_terminated_c1[c1_len] = null_terminated_c2[c2_len] =
  null_terminated_c3[c3_len] = '\0';
printf("d1=%.1f, f1=%.1f, i1=%d, i2=%lld, l1=%d, l2=%d, "
  "c1_len=%d, c2_len=%d, c3_len=%d\n",
  *d1, *f1, *i1, *i2, *l1, *l2, c1_len, c2_len, c3_len);
printf("c1='%s', c2='%s', c3='%s'\n",
  null_terminated_c1, null_terminated_c2, null_terminated_c3);
fflush(stdout); /* Flush output before switching languages */
call_fortran();
}
/* C function designed to be called from Fortran, passing
arguments by * value */
int c_value__(double d, float f, int i, long long i8) {
  printf("d=%.1f, f=%.1f, i=%d, i8=%lld\n", d, f, i, i8);
  fflush(stdout); /* Flush output before switching languages */
  return 4; /* Nonzero will be treated as ".true." by Fortran */
}
```

Here is the Fortran source code (`f_part.f90`):

```
program f_part
```

```
  implicit none
  ! Explicit interface is not required, but adds some error-checking
  interface
    subroutine c_reference(d1, f1, i1, i2, c1, l1, l2, c2, c3)
    doubleprecision d1
    real f1
    integer i1
    integer*8 i2
    character*(*) c1, c3
    character*4 c2
    logical l1, l2
  end subroutine c_reference
  logical function c_value(d, f, i, i8)
    doubleprecision d
    real f
    integer i
    integer*8 i8
  end function c_value
end interface
logical l
pointer (p_user, user)
character*32 user
integer*8 getlogin_nounderscore ! File decorate.txt maps this to
external getlogin_nounderscore ! "getlogin" without underscore
intrinsic char
! Demonstrate calling from Fortran a C function taking arguments by
! reference
call c_reference(9.8d0, 7.6, 5, 4_8, 'hello', .false., .true., &
  'from', 'f_part')
! Demonstrate calling from Fortran a C function taking arguments by ! value.
l = c_value(%val(9.8d0), %val(7.6), %val(5), %val(4_8))
write(6 , "(a,l8)") "l=", l
! "getlogin" is a standard C library function which returns "char *".
! When a C function returns a pointer, you must use a Cray pointer to
! receive the address and examine the data at that address, instead
! of assigning to an ordinary variable
p_user = getlogin_nounderscore()
write(6, "(3a)") "'", user(1:index(user, char(0)) - 1), "'"
end program f_part
! Subroutine to be called from C
subroutine f1(c, i, i8, f, d, l)
  implicit none
  intrinsic flush
  character*(*) c
  integer i
  integer*8 i8
  real f
  doubleprecision d
  logical l
  write(6, "(3a,2i5,2f5.1,l8)") "'", c, "'", i, i8, f, d, l
  call flush(6); ! Flush output before switching languages
end subroutine f1
```

And here is the third file (`decorate.txt`):

```
getlogin_nounderscore getlogin
```

Compile and execute these three files (`c_part.c`, `f_part.f90`, and `decorate.txt`) like this:

```
$ pathf90 -Wall -intrinsic=flush -fdecorate decorate.txt
    f_part.f90 c_part.c
$ ./a.out
d1=9.8, f1=7.6, i1=5, i2=4, l1=0, l2=1, c1_len=5, c2_len=4, c3_len=6
c1='hello', c2='from', c3='f_part'
'hello from call_fortran' 123 456 7.8 9.1 T
d=9.8, f=7.6, i=5, i8=4
l= T
'johndoe'
```

3.5.1.2 Example: Accessing common blocks from C

Variables in Fortran 90 modules are grouped into common blocks, one for initialized data and another for uninitialized data. It is possible to use `-fdecorate` to access these common blocks from C, as shown in this example:

```
$ cat mymodule.f90
module mymodule
  public
    integer :: modulevar1
    doubleprecision :: modulevar2
    integer :: modulevar3 = 44
    doubleprecision :: modulevar4 = 55.5
end module mymodule
program myprogram
  use mymodule
  modulevar1 = 22
  modulevar2 = 33.3
  call mycfunction()
end program myprogram
$ cat mycprogram.c
#include <stdio.h>
extern struct {
  int modulevar1;
  double modulevar2;
  } mymodule_data;
extern struct {
  int modulevar3;
  double modulevar4;
  } mymodule_data_init;
void mycfunction()
{
  printf("%d %g\n", mymodule_data.modulevar1, mymodule_data.modulevar2);
  printf("%d %g\n", mymodule_data_init.modulevar3,
    mymodule_data_init.modulevar4);
}
$ cat dfile
.data_init.in.mymodule mymodule_data_init
.data.in.mymodule.in.mymodule mymodule_data
mycfunction mycfunction
$ pathf90 -fdecorate dfile mymodule.f90 mycprogram.c
mymodule.f90:
mycprogram.c:
```

```
$ ./a.out
22 33.3
44 55.5
```

3.6 Runtime I/O compatibility

Files generated by the Fortran I/O libraries on other systems may contain data in different formats than that generated or expected by codes compiled by the PathScale EKOPath Fortran compiler. This section discusses how the PathScale EKOPath Fortran compiler interacts with files created by other systems.

3.6.1 Performing endian conversions

Use the assign command, or the `ASSIGN()` procedure, to perform endian conversions while doing file I/O.

3.6.1.1 The assign command

The `assign` command changes or displays the I/O processing directives for a Fortran file or unit. The `assign` command allows various processing directives to be associated with a unit or file name. This can be used to perform numeric conversion while doing file I/O.

The `assign` command uses the file pointed to by the `FILENV` environment variable to store the processing directives. This file is also used by the Fortran I/O libraries to load directives at runtime.

For example:

```
$ FILENV=.assign
$ export FILENV
$ assign -N mips u:15
```

This instructs the Fortran I/O library to treat all numeric data read from or written to unit 15 as being MIPS-formatted data. This effectively means that the contents of the file will be translated from big-endian format (MIPS) to little-endian format (Intel) while being read. Data written to the file will be translated from little-endian format to big-endian format.

See the `assign(1)` man page for more details and information.

3.6.1.2 Using the wildcard option

The wildcard option for the `assign` command is:

```
assign -N mips p:%
```

Before running your program, run the following commands:

```
$ FILENV=.assign
$ export FILENV
$ assign -N mips p:%
```

This example matches all files.

3.6.1.3 Converting data and record headers

To convert numeric data in all unformatted units from big endian, and convert the record headers from big endian, use the following:

```
$ assign -F f77.mips -N mips g:su
$ assign -I -F f77.mips -N mips g:du
```

The `su` specifier matches all sequential unformatted open requests. The `du` specifier matches all direct unformatted open requests. The `-F` option sets the record header format to big endian (`F77.mips`).

3.6.1.4 The ASSIGN() procedure

The `ASSIGN()` procedure provides a programmatic interface to the `assign` command. It takes as an argument a string specifying the `assign` command and an integer to store a returned error code. For example:

```
integer :: err
call ASSIGN("assign -N mips u:15", err)
```

This example has the same effect as the example in Section 3.6.1.1.

3.6.1.5 I/O compilation flags

Two compilation flags have been added to help with I/O: `-byteswapio` and `-convert conversion`.

The `-byteswapio` flag swaps bytes during I/O so that unformatted files on a little-endian processor are read and written in big-endian format (or vice versa.) The `-convert conversion` flag controls the swapping of bytes during I/O so that unformatted files on a little-endian processor are read and written in big-endian format (or vice versa.) To be effective, the option must be used when compiling the Fortran main program.

Setting the environment variable `FILENV` when running the program will override the compiled-in choice in favor of the choice established by the command `assign`. The `-convert conversion` flag can take one of three arguments:

- `native` - no conversion, the default
- `big_endian` - files are big-endian
- `little_endian` - files are little-endian

For more details, see the `pathf95` man page.

3.6.2 Reserved file units

The EKOPath Fortran compiler reserves Fortran file units 5, 6, and 0.

3.7 Source code compatibility

This section discusses our compatibility with source code developed for other compilers. Different compilers represent types in various ways, and this may cause some problems.

3.7.1 Fortran KINDs

The Fortran `KIND` attribute is a way to specify the precision or size of a type. Modern Fortran uses `KINDS` to declare types. This system is very flexible, but has one drawback. The recommended and portable way to use `KINDS` is to find out what they are like this:

```
integer :: dp_kind = kind(0.0d0)
```

In actuality, some users hard-wire the actual values into their programs:

```
integer :: dp_kind = 8
```

This is an unportable practice, because some compilers use different values for the `KIND` of a double-precision floating point value.

The majority of compilers use the number of bytes in the type as the `KIND` value. For floating point numbers, this means `KIND=4` is 32-bit floating point, and `KIND=8` is 64-bit floating point. The PathScale compiler follows this convention.

Unfortunately for us and our users, this is incompatible with unportable programs written using GNU Fortran, `g77`. `g77` uses `KIND=1` for single precision (32 bits) and `KIND=2` for double precision (64 bits). For integers, however, `g77` uses `KIND=3` for 1 byte, `KIND=5` for 2 bytes, `KIND=1` for 4 bytes, and `KIND=2` for 8 bytes.

We are investigating the cost of providing a compatibility flag for unportable `g77` programs. If you find this to be a problem, the best solution is to change your program to inquire for the actual `KIND` values instead of hard-wiring them.

If you are using `-i8` or `-r8`, see Section 3.3.1 for more details on usage.

3.8 Library compatibility

This section discusses our compatibility with libraries compiled with C or other Fortran compilers.

Linking object code compiled with other Fortran compilers is a complex issue. Fortran 90 or 95 compilers implement modules and arrays so differently that it is extremely difficult to attempt to link code from two or more compilers. For Fortran 77, run-time libraries for things like I/O and intrinsics are different, but it is possible to link both runtime libraries to an executable.

We have experimented using object code compiled by `g77`. This code is not guaranteed to work in every instance. It is possible that some of our library functions have the same name but different calling conventions than some of `g77`'s library functions. We have not tested linking object code from other compilers, with the exception of `g77`.

3.8.1 Name mangling

Name mangling is a mechanism by which names of functions, procedures, and common blocks from Fortran source files are converted into an internal representation when compiled into object files. For example, a Fortran subroutine called `foo` gets turned into the name "`foo_`" when placed in the object file. We do this to avoid name collisions with similar functions in other libraries. This makes mixing code from C, C++, and Fortran easier.

Name mangling ensures that function, subroutine, and common-block names from a Fortran program or library do not clash with names in libraries from other programming languages. For example, the Fortran library contains a function named "`access`", which performs the same function as the function `access` in the standard C library. However, the Fortran library `access` function takes four arguments, making it incompatible with the standard C library `access` function, which takes only two arguments. If your program links with the standard C library, this would cause a symbol name clash. Mangling the Fortran symbols prevents this from happening.

By default, we follow the same name mangling conventions as the GNU `g77` compiler and `libf2c` library when generating mangled names. Names without an underscore have a single underscore appended to them, and names containing an underscore have two underscores appended to them. The following examples should help make this clear:

```
molecule -> molecule_
run_check -> run_check__
energy_ -> energy___
```

This behavior can be modified by using the `-fno-second-underscore` and the `-fno-underscoring` options to the `pathf95` compiler.

PGI Fortran and Intel Fortran's default policies correspond to our `-fno-second-underscore` option.

Common block names are also mangled. Our name for the blank common block is the same as `g77` (`_BLNK__`). PGI's compiler uses the same name for the blank common block, while Intel's compiler uses `_BLANK__`.

3.8.2 ABI compatibility

The PathScale EKOPath compilers support the official x86_64 Application Binary Interface (ABI), which is not always followed by other compilers. In particular, `g77` does not pass the return values from functions returning `COMPLEX` or `REAL` values according to the x86_64 ABI. (Double precision `REAL`s are OK.) For more details about what `g77` does, see the "`info g77`" entry for the `-ff2c` flag.

This issue is a problem when linking binary-only libraries such as Kazushige Goto's `BLAS` library or the `ACML` library (AMD Core Math Library)[1]. Libraries such as `FFTW` and `MPICH` don't have any functions returning `REAL` or `COMPLEX`, so there are no issues with these libraries.

For linking with `g77`-compiled functions returning `COMPLEX` or `REAL` values see Section 3.8.3.

Like most Fortran compilers, we represent character strings passed to subprograms with a character pointer, and add an integer length parameter to the end of the call list.

3.8.3 Linking with g77-compiled libraries

If you wish to link with a library compiled by `g77`, and if that library contains functions that return `COMPLEX` or `REAL` types, you need to tell the PathScale compiler to treat those functions differently.

Use the `-ff2c-abi` switch at compile time to point the PathScale compiler at a file that contains a list of functions in the `g77`-compiled libraries that return `COMPLEX` or `REAL` types. When the PathScale compiler generates code that calls these listed functions, it will modify its ABI behavior to match `g77`'s expectations. The `-ff2c-abi` flag is used at compile time and not at link time.

NOTE: You can only specify the `-ff2c-abi` switch once on the command line. If you have multiple `g77`-compiled libraries, you need to place all the appropriate symbol names into a single file.

The format of the file is one symbol per line. Each symbol should be as you would specify it in your Fortran code (i.e. do not mangle the symbol). As an example:

```
$ cat example-list
sdot
cdot
$
```

You can use the `fsymlist` program to generate a file in the appropriate format. For example:

```
$ fsymlist /opt/gnu64/lib/mylibrary.a > mylibrary-list
```

[1] We have not tested `ACML` on the EM64T version of the compiler suite.

This will find all Fortran symbols in the `mylibrary.a` library and place them into the `mylibrary-2.0-list` file. You can then use this file with the `-ff2c-abi` switch.

NOTE: The `fsymlist` program generates a list of *all* Fortran symbols in the library, including those that do not return `COMPLEX` or `REAL` types. The extra symbols will be ignored by the compiler.

3.8.3.1 AMD Core Math Library (ACML)

The AMD Core Math Library (`ACML`) incorporates `BLAS`, `LAPACK`, and `FFT` routines, and is designed to obtain maximum performance from applications running on AMD platforms. This highly optimized library contains numeric functions for mathematical, engineering, scientific, and financial applications. `ACML` is available both as a 32-bit library (for compatibility with legacy x86 applications), and as a 64-bit library that is designed to fully exploit the large memory space and improved performance offered by the x86_64 architecture[2].

To use `ACML 1.5` with the PathScale EKOPath Fortran compiler, use the following:

```
$ pathf95
    foo.f bar.f -lacml
```

To use `ACML 2.0` with the PathScale EKOPath Fortran compiler, use the following:

```
$ pathf95 -L<path_to_acml_lib>
    foo.f bar.f -lacml
```

ACML 2.5.1 and later, built with the PathScale compilers, is available from the AMD website at `http://developer.amd.com/acml.aspx`. With these later versions of ACML, the workarounds described above are unnecessary.

3.8.4 List directed I/O and repeat factors

By default, when list directed I/O is used and two or more consecutive values are identical, the output uses a repeat factor.

For example:

```
real :: a(5)=88.0
write (*,*) a
end
```

This example generates the following output:

```
5*88.
```

[2]We have not tested `ACML` on the EM64T version of the compiler suite.

This behavior conforms to the language standard. However, some users prefer to see multiple values instead of the repeat factor:

```
88., 88., 88., 88., 88.
```

There are two ways to accomplish this, using an environment variable and using the assign command.

3.8.4.1 Environment variable

If the environment variable `FTN_SUPPRESS_REPEATS` is set before the program starts executing, then list-directed "write" and "print" statements will output multiple values instead of using the repeat factor.

To output multiple values when running within the `bash` shell:

```
export FTN_SUPPRESS_REPEATS=yes
```

To output multiple values when running within the `csh` shell:

```
setenv FTN_SUPPRESS_REPEATS yes
```

To output repeat factors when running within the `bash` shell:

```
unset FTN_SUPPRESS_REPEATS
```

To output repeat factors when running within the `csh` shell:

```
unsetenv FTN_SUPPRESS_REPEATS
```

3.8.4.2 Assign command

Using the `-y on` option to the `assign` command will cause all list directed output to the specified file names or unit numbers to output multiple values; using the `-y off` option will cause them to use repeat factors instead.

For example, to output multiple values on logical unit 6 and on any logical unit which is associated with file `test2559.out`, type these commands before running the program:

```
export FILENV=myassignfile
assign -I -y on u:6
assign -I -y on f:test2559.out
```

The following program would then use no repeat factors, because the first `write` statement refers explicitly to unit 6, the second `write` statement refers implicitly to unit 6 (by using "*" in place of a logical unit), and the third is bound to file `test2559.out`:

```
real :: a(5)=88.0
write (6,*) a
write (*,*) 77.0, 77.0, 77.0, 77.0, 77.0
open(unit=17, file='test2559.out')
write (17,*) 99.0, 99.0, 99.0, 99.0, 99.0
end
```

3.9 Porting Fortran code

The following option can help you fix problems prior to porting your code.

`-r8 -i8` Respectively promotes the default representation for `REAL` and `INTEGER` type from 4 bytes to 8 bytes. Useful for porting from Cray code when integer and floating point data is 8 bytes long by default. Watch out for type mismatches with external libraries.

These sections contain helpful information for porting Fortran code:

- Section 3.7.1 has information on porting code that includes `KINDS`, sometimes a problem when porting Fortran code
- Section 3.7 has information on source code compatibility
- Section 3.8 has information on library compatibility

3.10 Debugging and troubleshooting Fortran

The flag `-g` tells the PathScale EKOPath compilers to produce data in the form used by modern debuggers, such as PathScale's `pathdb`, GDB, Etnus' TotalView®, Absoft Fx2™, and Streamline's DDT™. This format is known as DWARF 2.0 and is incorporated directly into the object files. Code that has been compiled using `-g` will be capable of being debugged using `pathdb`, GDB, or other debuggers.

The `-g` option automatically sets the optimization level to `-O0` unless an explicit optimization level is provided on the command line. Debugging of higher levels of optimization is possible, but the code transforming performed by the optimizations many make it more difficult.

Bounds checking is quite a useful debugging aid. This can also be used to debug allocated memory.

If you are noticing numerical accuracy problems, see Section 7.7 for more information on numerical accuracy.

See Section 10 for more information on debugging and troubleshooting. See the *PathScale Debugger User Guide* for more information on `pathdb`.

3.10.1 Writing to constants can cause crashes

Some Fortran compilers allocate storage for constant values in read-write memory. The PathScale EKOPath Fortran compiler allocates storage for constant values in read-only memory. Both strategies are valid, but the PathScale compiler's approach allows it to propagate constant values aggressively.

This difference in constant handling can result in crashes at runtime when Fortran programs that write to constant variables are compiled with the PathScale EKOPath Fortran compiler. A typical situation is that an argument to a subroutine or function is given a constant value such as `0` or `.FALSE.`, but the subroutine or function tries to assign a new value to that argument.

We recommend that where possible, you fix code that assigns to constants so that it no longer does this. Such a change will continue to work with other Fortran compilers, but will allow the PathScale EKOPath Fortran compiler to generate code that will not crash and will run more efficiently.

If you cannot modify your code, we provide an option called `-LANG:rw_const=on` that will change the compiler's behavior so that it allocates constant values in read-write memory. We do not make this option the default, as it reduces the compiler's ability to propagate constant values, which makes the resulting executables slower.

You might also try the `-LANG:formal_deref_unsafe` option. This option tells the compiler whether it is unsafe to speculate a dereference of a formal parameter in Fortran. The default is `OFF`, which is better for performance. See the `eko` man page for more details on these two flags.

3.10.2 Runtime errors caused by aliasing among Fortran dummy arguments

The Fortran standards require that arguments to functions and subroutines not alias each other. As an example, this is illegal:

```
program bar
...
call foo(c,c)
...
subroutine foo(a,b)
integer i
real a(100), b(100)
do i = 2, 100
  a(i) = b(i) - b(i-1)
enddo
```

Because `a` and `b` are dummy arguments, the compiler relies on the assumption that `a` and `b` are in non-overlapping areas of memory when it optimizes the program. The resulting program when run will give wrong results.

Programmers occasionally break this aliasing rule, and as a result, their programs get the wrong answer only under high levels of optimization. This sort

of bug frequently is thought to be a compiler bug, so we have added this option to the compiler for testing purposes. If your failing program gets the right answer with `-OPT:alias=no_parm` or `-WOPT:fold=off`, then it is likely that your program is breaking this Fortran aliasing rule.

3.10.3 Fortran malloc debugging

The PathScale EKOPath Compiler Suite includes a feature to debug Fortran memory allocations. By setting the environment variable `PSC_FDEBUG_ALLOC`, memory allocations will be initialized during execution to the following values:

```
PSC_FDEBUG_ALLOC   Value
----------------------
ZERO               0
NaN                0xffa5a5a5 (4 byte NaN)
NaN8               0xffa5a5a5fff5a5a5ll (8 byte NaN)
```

For example, to initialize all memory allocations to zeroes, set `PSC_FDEBUG_ALLOC=ZERO` before running the program. The four-byte and eight-byte NaNs will only initialize arrays that are aligned with their width (32 and 64 bits, respectively).

3.11 Fortran compiler stack size

The Fortran compiler allocates data on the stack by default. Some environments set a low limit on the size of a process's stack, which may cause Fortran programs that use a large amount of data to crash shortly after they start.

If the PathScale EKOPath Fortran runtime environment detects a low stack size limit, it will automatically increase the size of the stack allocated to a Fortran process before the Fortran program begins executing.

By default, it automatically increases this limit to the total amount of physical memory on a system, less 128 megabytes per CPU. For example, when run on a 4-CPU system with 1G of memory, the Fortran runtime will attempt to raise the stack size limit to 1G - (128M * 4), or 640M.

To have the Fortran runtime tell you what it is doing with the stack size limit, set the `PSC_STACK_VERBOSE` environment variable before you run a Fortran program. You can control the stack size limit that the Fortran runtime attempts to use using the `PSC_STACK_LIMIT` environment variable.

If this is set to the empty string, the Fortran runtime will not attempt modify the stack size limit in any way.

Otherwise, this variable must contain a number. If the number is not followed by any text, it is treated as a number of bytes. If it is followed by the letter "`k`" or "`K`", it is treated as kilobytes (1024 bytes). If "`m`" or "`M`", it is treated as megabytes (1024K). If "`g`" or "`G`", it is treated as gigabytes (1024M). If "%", it is treated as a percentage of the system's physical memory.

If the number is negative, it is treated as the amount of memory to leave free, i.e. it is subtracted from the amount of physical memory on the machine. If all of this text is followed by "/cpu", it is treated as a "per cpu" number, and that number is multiplied by the number of CPUs on the system. This is useful for multiprocessor systems that are running several processes concurrently. The value specified (implicitly or explicitly) is the memory value per process.

Here are some sample stack size settings (on a 4 CPU system with 1G of memory):

Value	**Meaning**
100000	100000 bytes
820K	820K (839680 bytes)
-0.25g	all but 0.25G, or 0.75G total
128M/cpu	128M per CPU, or 512M total
-10M/cpu	all but 10M per CPU (all but 40M total), or 0.96G total

If the Fortran runtime encounters problems while attempting to modify the stack size limit, it will print some warning messages, but will not abort.

Chapter 4

The PathScale EKOPath C/C++ compiler

The PathScale EKOPath C and C++ compilers conform to the following set of standards and extensions.

The C compiler:

- Conforms to ISO/IEC 9899:1990, Programming Languages - C standard
- Supports extensions to the C programming language as documented in "*Using GCC: The GNU Compiler Collection Reference Manual*," October 2003, for GCC version 3.3.1
- Refer to Section 4.4 of this document for the list of extensions that are currently not supported
- Complies with the C Application Binary Interface as defined by the GNU C compiler (`gcc`) as implemented on the platforms supported by the PathScale EKOPath Compiler Suite
- Supports most of the widely used command-line options supported by `gcc`
- Generates code that complies with the x86_64 ABI and the 32-bit x86 ABI

The C++ compiler:

- Conforms to ISO/IEC 14882:1998(E), Programming Languages - C++ standard
- Supports extensions to the C++ programming language as documented in "*Using GCC: The GNU Compiler Collection Reference Manual*," October 2003, for GCC version 3.3.1
- Refer to Section 4.4 of this document for the list of extensions that are currently not supported
- Complies with the C Application Binary Interface as defined by the GNU C++ compiler (`g++`) as implemented on the platforms supported by the PathScale EKOPath Compiler Suite

- Supports most of the widely used command-line options supported by `g++`
- Generates code that complies with the x86_64 ABI and the 32-bit x86 ABI

To invoke the PathScale EKOPath C and C++ compilers, use these commands:

- `pathcc` - invoke the C compiler
- `pathCC` - invoke the C++ compiler

The command-line flags for both compilers are compatible with those taken by the GCC suite. See Section 4.1 for more discussion of this.

4.1 Using the C/C++ compilers

If you currently use the GCC compilers, the PathScale EKOPath compiler commands will be familiar. Makefiles that presently work with GCC should operate with the PathScale EKOPath compilers effortlessly–simply change the command used to invoke the compiler and rebuild. See Section **??** for information on modifying existing scripts

The invocation of the compiler is identical to the GCC compilers, but the flags to control the compilation are different. We have sought to provide flags compatible with GCC's flag usage whenever possible and also provide optimization features that are absent in GCC, such as IPA and LNO.

Generally speaking, instead of being a single component as in GCC, the PathScale compiler is structured into components that perform different classes of optimizations. Accordingly, compilation flags are provided under group names like `-IPA`, `-LNO`, `-OPT`, `-CG`, etc. For this reason, many of the compilation flags in PathScale will differ from those in GCC. See the `eko` man page for more information.

The default optimization level is 2. This is equivalent to passing `-O2` as a flag. The following three commands are identical in their function:

```
$ pathcc hello.c
$ pathcc -O hello.c
$ pathcc -O2 hello.c
```

See Section 7.1 for information about the optimization levels available for use with the compiler.

To run with `-Ofast` or with `-ipa`, the flag must also be given on the link command.

```
$ pathCC -c -Ofast warpengine.cc
$ pathCC -c -Ofast wormhole.cc
$ pathCC -o ftl -Ofast warpengine.o wormhole.o
```

See Section 7.3 for information on `-ipa` and `-Ofast`.

4.2 Compiler and runtime features

4.2.1 Preprocessing source files

Before being passed to the compiler front-end, source files are optionally passed through a source code preprocessor. The preprocessor searches for certain directives in the file and, based on these directives, can include or exclude parts of the source code, include other files, or define and expand macros.

All C and C++ files are passed through the the C preprocessor unless the `-noccp` flag is specified.

4.2.1.1 Pre-defined macros

The PathScale compiler pre-defines some macros for preprocessing code. These include the following:

```
__linux
__linux__
linux
__unix
__unix__
unix
__gnu_linux__
__GNUC__ 3
__GNUC_MINOR__ 3
__GNUC_PATCHLEVEL__ 1
__PATHSCALE__ "2.0"
__PATHCC__ 2
__PATHCC_MINOR__ 0
__PATHCC_PATCHLEVEL__ 0
```

NOTE: The `__GNU*` and `__PATH*` values are derived from the respective compiler version numbers, and will change with each release.

These Fortran macros will also used if the source file is Fortran, but `cpp` is used.

```
_LANGUAGE_FORTRAN 1
LANGUAGE_FORTRAN 1
_LANGUAGE_FORTRAN90 1
LANGUAGE_FORTRAN90 1
```

For 32-bit compilation, the following macros are defined:

```
__i386
__i386__
i386
```

For 64-bit, the following macros are defined:

```
__LP64__
_LP64
```

NOTE: When using an optimization level at `-O1` or higher, the compiler will use the `__OPTIMIZE__` macro.

A quick way to list all the predefined `cpp` macros would be to compile your program with the flags `-dD -keep`. You can find all the defines (or predefined macros) in the resulting `.i` file. Here is an example for C:

```
$ cat hello.c
main(){
printf("Hello World\n");
}
$ pathcc -dD -keep hello.c
$
$ wc hello.i
  94 278 2606 hello.i
$ cat hello.i
```

The `hello.i` file will contain the list of pre-defined macros.

NOTE: Generating an `.i` file doesn't work well with Fortran, because if the preprocessor sends the "`#define`"s to the `.i` file, Fortran can't parse them. See Section 3.4.3.1 for information on finding pre-defined macros in Fortran.

4.2.2 Pragmas

4.2.2.1 Pragma pack

In this release, we have tested and verified that the pragma `pack` is supported. The syntax for this pragma is:

`#pragma pack(n)` This pragma specifies that the next structure should have each of their fields aligned to an alignment of `n` bytes if its natural alignment is not smaller than `n`.

4.2.2.2 Changing optimization using pragmas

Optimization flags can now be changed via directives in the user program.

In C and C++, the directive is of the form:

```
#pragma options <list-of-options>
```

Any number of these can be specified inside function scopes. Each affects only the optimization of the entire function in which it is specified. The literal string can also contain an unlimited number of different options separated by space. The compilation of the next function reverts back to the settings specified in the compiler command line.

In this release, there are limitations to the options that are processed in this options directive, and their effects on the optimization.

- There is no warning or error given for options that are not processed.
- These directives are processed only in the optimizing backend. Thus, only options that affect optimizations are processed.
- In addition, it will not affect the phase invocation of the backend components. For example, specifying `-O0` will not suppress the invocation of the global optimizer, though the invoked backend phases will honor the specified optimization level.
- Apart from the optimization level flags, only flags belonging to the following option groups are processed: `-LNO`, `-OPT` and `-WOPT`.

4.2.2.3 Code layout optimization using pragmas

This pragma is applicable to C/C++. The user can provide a hint to the compiler regarding which branch of an IF-statement is more likely to be executed at runtime. This hint allows the compiler to optimize code generated for the different branches.

The directive is of the form:

```
#pragma frequency_hint <hint>
```

where <*hint*> is a choice from:

- `never` : The branch is rarely or never executed.
- `init` : The branch is executed only during initialization.
- `frequent` : The branch is executed frequently.

The branch of the IF-statement that contains the pragma will be affected.

4.2.3 Mixing code

If you have a large application that mixes Fortran code with code written in other languages, and the main entry point to your application is from C or C++, you can optionally use `pathcc` or `pathCC` to link the application, instead of `pathf95`. If you do, you must manually add the Fortran runtime libraries to the link line.

See Section 3.5 for details. To link object files that were generated with `pathCC` using `pathcc` or `pathf95`, include the option `-lstdc++`.

4.2.4 Linking

Note that the `pathcc` (C language) user needs to add `-lm` to the link line when calling `libm` functions. The second pass of feedback compilation may require an explicit `-lm`.

4.3 Debugging and troubleshooting C/C++

The flag `-g` tells the PathScale EKOPath C and C++ compilers to produce data in the form used by modern debuggers, such as `pathdb` or GDB. This format is known as DWARF 2.0 and is incorporated directly into the object files. Code that has been compiled using `-g` will be capable of being debugged using `pathdb`, GDB, or other debuggers.

The `-g` option automatically sets the optimization level to `-O0` unless an explicit optimization level is provided on the command line. Debugging of higher levels of optimization is possible, but the code transformation performed by the optimizations may make it more difficult.

See Section 10 for more information on troubleshooting and debugging. See the *PathScale Debugger User Guide* for more information on `pathdb`.

4.4 GCC extensions not supported

The PathScale EKOPath C and C++ Compiler Suite supports most of the C and C++ extensions supported by GCC Version 3.3.1 Suite. In this release, we do not support the following extensions:

For C:

- Nested functions
- Complex integer data type: Complex integer data types are not supported. Although the PathScale EKOPath Compiler Suite fully supports floating point complex numbers, it does not support complex integer data types, such as `_Complex int`.
- Thread local storage
- SSE3 intrinsics
- Many of the `__builtin` functions
- A `goto` outside of the block. PathScale compilers do support taking the address of a label in the current function and doing indirect jumps to it.
- The compiler generates incorrect code for structs generated on the fly (a GCC extension).

For C++:

- Java-style exceptions
- `java_interface` attribute
- `init_priority` attribute

Chapter 5

Porting and compatibility

5.1 Getting started

Here are some tips to get you started compiling selected applications with the PathScale EKOPath Compiler Suite.

5.2 GNU compatibility

The PathScale EKOPath Compiler Suite C, C++, and Fortran compilers are compatible with `gcc` and `g77`. Some packages will check strings like the `gcc` version or the name of the compiler to make sure you are using `gcc`; you may have to work around these tests. See Section 5.6.1 for more information.

Some packages continue to use deprecated features of `gcc`. While `gcc` may print a warning and continue compilation, the PathScale EKOPath Compiler Suite C, C++, and Fortran compilers may print an error and exit. Use the instructions in the error to substitute an updated flag. For example, some packages will specify the deprecated "`-Xlinker`" `gcc` flag to pass arguments to the linker, while the PathScale EKOPath Compiler Suite uses the modern `-Wl` flag.

Some `gcc` flags may not yet be implemented. These will be documented in the release notes.

If a `configure` script is being used, PathScale provides wrapper scripts for `gcc` that are frequently helpful. See Section 5.6.1 for more information.

5.3 Porting Fortran

If you are porting Fortran code, see Section 3.9 for more information about Fortran-specific issues.

5.3.1 Intrinsics

The PathScale Fortran compiler supports many intrinsics and also has many unique intrinsics of its own. See Appendix C for the complete list of supported intrinsics.

5.3.1.1 An example

Here is some sample output from compiling Amber 8 using only `ANSI` intrinsics. You get this series of error messages:

```
$ pathf95 -O3 -msse2 -m32 -o fantasian
fantasian.o ../../lib/random.o ../../lib/mexit.o
fantasian.o: In function ‘simplexrun_’:
fantasian.o(.text+0xaad4): undefined reference to ‘rand_’
fantasian.o(.text+0xab0e): undefined reference to ‘rand_’
fantasian.o(.text+0xab48): undefined reference to ‘rand_’
fantasian.o(.text+0xab82): undefined reference to ‘rand_’
fantasian.o(.text+0xabbf): undefined reference to ‘rand_’
fantasian.o(.text+0xee0a): more undefined references to ‘rand_’ follow
collect2: ld returned 1 exit status
```

The problem is that `RAND` is not `ANSI`. The solution is to build the code with the flag `-intrinsic=PGI`.

5.3.2 Name-mangling

Name mangling ensures that function, subroutine, and common-block names from a Fortran program or library do not clash with names in libraries from other programming languages. This makes mixing code from C, C++, and Fortran easier. See Section 3.8.1 for details on name mangling.

5.3.3 Static data

Some codes expect data to be initialized to zero and allocated in the heap. If this is the case with your code use the `-static` flag when compiling.

5.4 Porting to x86_64

Keep these things in mind when porting existing code to x86_64:

- Some source packages make assumptions about the locations of libraries and fail to look in `lib64`-named directories for libraries resulting in unresolved symbols at during the link.
- For the x86 platform, use the `-mcpu` flag `x86any` to specify the x86 platform, like this: `-mcpu=x86_64`.

5.5 Migrating from other compilers

Here is a suggested step-by-step approach to migrating code from other compilers to the PathScale EKOPath compilers:

1. Check the compiler name in your makefile; is the correct compiler being called?
   For example, you may need to add a line like this:
   `$ CC=pathcc ./configure <options>`
   Change the compiler in your makefile to `pathcc` or `pathf95`.
2. Check any flags that are called to be sure that the PathScale EKOPath Compiler Suite supports them. See the `eko` man page in Appendix E for a complete listing of supported flags.
3. If you plan on using IPA, see Section 7.3 for suggestions.
4. Compile your code and look at the results.
   (a) Did the program compile and link correctly? Are there missing libraries that were previously linked automatically?
   (b) Look for behavior differences; does the program behave correctly? Are you getting the right answer (for example, with numerical analysis)?

5.6 Compatibility

5.6.1 GCC compatibility wrapper script

Many software build packages check for the existence of `gcc`, and may even require the compiler used to be called `gcc` in order to build correctly. To provide complete compatibility with `gcc`, we provide a set of GCC compatibility wrapper scripts in `/opt/pathscale/compat-gcc/bin` (or `<install_directory>/compat-gcc/bin`).

This script can be invoked with different names:

- `gcc`, `cc` - to look like the GNU C compiler, and call `pathcc`
- `g++`, `c++` - to look like the GNU C++ compiler, and call `pathCC`
- `g77`, `f77` - to look like the GNU Fortran compiler, and call `pathf95`

To use this script, you must put the path to this directory in your shell's search path *before* the location of your system's `gcc` (which is usually `/usr/bin`). You can confirm the order in the search path by running "`which gcc`" after modifying your search path. The output should print the location of the gcc wrapper, not `/usr/bin/gcc`.

Chapter 6

Tuning Quick Reference

This chapter provides some ideas for tuning your code's performance with the PathScale EKOPath compiler.

The following sections describe a small set of tuning options that are relatively easy to try, and often give good results. These are tuning options that do not require Makefile changes, or risk the correctness of your code results. More detail on these flags can be found in the next chapter and in the man pages. A comprehensive list of the options for the PathScale EKOPath compiler can be found in the `eko` man page.

6.1 Basic optimization

Here are some things to try first when optimizing your code.

The basic optimization flag `-O` is equivalent to `-O2`. This is the first flag to think about using when tuning your code. Try:

```
  -O2
then ,
  -O3
and then,
  -O3 -OPT:Ofast.
```

For more information on the `-O` flags and `-OPT:Ofast`, see Section 7.1.

6.2 IPA

Inter-Procedural Analysis (IPA), invoked most simply with `-ipa`, is a compilation technique that analyzes an entire program. This allows the compiler to do optimizations without regard to which source file the code appears in. IPA can improve performance significantly.

IPA can be used in combination with the other optimization flags. `-O3 -ipa` or `-O2 -ipa` will typically provide increased performance over the `-O3` or `-O2` flags alone. `-ipa` needs to be used both in the compile and in the link steps of a build. See Section 7.3 for more details on how to use `-ipa`.

6.3 Feedback Directed Optimization (FDO)

Feedback directed optimization uses a special instrumented executable to collect profile information about the program that is then used in later compilations to tune the executable.

See Section 7.6 for more information.

6.4 Aggressive optimization

The PathScale EKOPath compilers provide an extensive set of additional options to cover special case optimizations. The ones documented in Chapter 7 contain options that may significantly improve the speed or performance of your code.

This section briefly introduces some of the first tuning flags to try beyond `-O2` or `-O3`. Some of these options require knowledge of what the algorithms are and what coding style of the program require, otherwise they may impact the program's correctness. Some of these options depend on certain coding practices to be effective.

One word of caution: The PathScale EKOPath Compiler Suite, like all modern compilers, has a range of optimizations. Some produce identical program output to the non-optimized, some can change the program's behavior slightly. The first class of optimizations is termed "safe" and the second "unsafe". See for Section 7.7 for more information on these optimizations.

`-OPT:Olimit=0` is a generally safe option but may result in the compilation taking a long time or consuming large quantities of memory. This option tells the compiler to optimize the files being compiled at the specified levels no matter how large they are.

The option `-fno-math-errno` bypasses the setting of ERRNO in math functions. This can result in a performance improvement if the program does not rely on IEEE exception handling to detect runtime floating point errors.

`-OPT:roundoff=2` also allows for fairly extensive code transformations that may result in floating point round-off or overflow differences in computations. Refer to Section 7.7.4.2 and 7.7.4 for more information.

The option `-OPT:div_split=ON` allows the conversion of `x/y` into `x*(recip(y)),` which may result in less accurate floating point computations. Refer to Sections 7.7.4.2 and 7.7.4 for more information.

The `-OPT:alias` settings allow the compiler to apply more aggressive optimizations to the program. The option `-OPT:alias=typed` assumes that the program has been coded in adherence with the ANSI/ISO C standard, which states that two pointers of different types cannot point to the same location in memory. Setting `-OPT:alias=restrict` allows the compiler to assume that points refer to distinct, non-overlapping objects. If the these options are specified and the program does violate the assumptions being made, the program may behave incorrectly. Refer to Section 7.7.1 for more information.

There are several shorthand options that can be used in place of the above options. The option `-OPT:Ofast` is equivalent to `-OPT:roundoff=2:Olimit=0:div_split=ON:alias=typed`. `-Ofast` is equivalent to `-O3 -ipa -OPT:Ofast -fno-math-errno`. When using this shorthand options, make sure the impact of the option is understood by stepwise building up the functionality by using the equivalent options.

There are many more options that may help the performance of the program. These options are discussed elsewhere in the User Guide and in the associated man pages.

6.5 Performance analysis

In addition to these suggestions for optimizing your code, here are some other ideas to assist you in tuning. Section 2.11 discusses figuring out where to tune your code, using `time` to get an overview of your code, and using `pathprof` to find your program's hot spots.

6.6 Optimize your hardware

Make sure you are optimizing your hardware as well. Section 7.8 discusses getting the best performance out of x86_64-based hardware (Opteron, Athlon™64, Athlon™64 FX, and Intel®EM64T). Hardware configuration can have a significant effect on the performance of your application.

Chapter 7

Tuning options

This chapter discusses in more depth some of the major groups of flags available in the PathScale EKOPath Compiler Suite.

7.1 Basic optimizations: The `-O` flag

The `-O` flag is the first flag to think about using. See Table 7.3 showing the default flag settings for various levels of optimization.

`-O0` (O followed by a zero) specifies no optimization–this is useful for debugging. The `-g` debugging flag is fully compatible with this level of optimization.

NOTE: Using `-g` by itself without specifying `-O` will change the default optimization level from `-O2` to `-O0` unless explicitly specified.

`-O1` specifies minimal optimizations with no noticeable impact on compilation time compared with `-O0`. Such optimizations are limited to those applied within straight-line code (basic blocks), like peephole optimizations and instruction scheduling. The `-O1` level of optimization minimizes compile time.

`-O2` only turns on optimizations which always increase performance and the increased compile time (compared to `-O1`) is commensurate with the increased performance. This is the default, if you don't use any of the `-O` flags. The optimizations performed at level 2 are:

- For inner loops, perform:
  - Loop unrolling
  - Simple if-conversion
  - Recurrence-related optimizations
- Two passes of instruction scheduling
- Global register allocation based on first scheduling pass
- Global optimizations within function scopes:
  - Partial redundancy elimination
  - Strength reduction and loop termination test replacement
  - Dead store elimination
  - Control flow optimizations
  - Instruction scheduling across basic blocks
- `-O2` implies the flag `-OPT:goto=on,` which enables the conversion of `GOTO`s into higher level structures like `FOR` loops.
- `-O2` also sets `-OPT:Olimit=6000`

`-O3` turns on additional optimizations which will most likely speed your program up, but may, in rare cases, slow your program down. The optimizations provided at this level include all `-O1` and `-O2` optimizations and the flags noted below:

- `-LNO:opt=1` Turn on Loop Nest Optimization (for more details, see Section 7.4)
- `-OPT` with the following options in the OPT group: (see the `-opt` man pages for more information)

  `-OPT:roundoff=1` (see Section 7.7.4.2)

  `-OPT:IEEE_arith=2` (see Section 7.7.4)

  `-OPT:Olimit=9000` (see Section 6.3)

  `-OPT:reorg_common=1` (see the `eko(7)` man page)

NOTE: In our in-house testing, we have noticed that several codes which are slower at `-O3` than `-O2` are fixed by using `-O3 -LNO:prefetch=0`. This seems to mainly help codes that fit in cache.

7.2 Syntax for complex optimizations (`-CG, -IPA, -LNO -OPT, -WOPT`)

The group optimizations control a variety of behaviors and can override defaults. This section covers the syntax of these options.

The group options allow for the setting of multiple sub-options in two ways:

- Separating each sub-flag by colons, or
- Using multiple flags on the command line.

For example, the following command lines are equivalent:

```
pathcc -OPT:roundoff=2:alias=restrict wh.c
pathcc -OPT:roundoff=2 -OPT:alias=restrict wh.c
```

Some sub-options either enable or disable the feature. To enable a feature, either specify only the subflag name or with `=1`, `=ON`, or `=TRUE`. Disabling a feature, is accomplished by adding `=0`, `=OFF`, or `=FALSE`. The following command lines mean the same thing:

```
pathf95 -OPT:div_split:fast_complex=FALSE:IEEE_NaN_inf=OFF wh.F
pathf95 -OPT:div_split=1:fast_complex=0:IEEE_NaN_inf=false wh.F
```

7.3 Inter-Procedural Analysis (IPA)

Software applications are normally written and organized into multiple source files that make up the program. The compilation process, usually defined by a *Makefile*, invokes the compiler to compile each source file, called *compilation unit*, separately. This traditional build process is called *separate compilation*. After all compilation units have been compiled into `.o` files, the linker is invoked to produce the final executable.

The problem with separate compilation is that it does not provide the compiler with complete program information. The compiler has to make worst-case assumptions at places in the program that access external data or call external functions. In whole program optimization, the compiler can collect information over the entire program so it can make better decision on whether it is safe to perform various optimizations. Thus, the same optimization performed under whole program compilation will become much more effective. In addition, more types of optimization can be performed under whole program compilation than separate compilation.

This section presents the compilation model that enables whole program optimization in the PathScale EKOPath compiler and how it relates to the `-ipa` flag that invokes it at the user level. Various analyses and optimizations performed by IPA are described. How IPA improves the quality of the backend optimization is also explained. Various IPA-related flags that can be used to tune for program performance are presented and described. Finally, we have an example of the difference that IPA makes in the performance of the SPEC CPU2000 benchmark suite.

7.3.1 The IPA compilation model

Inter-procedural compilation is the mechanism that enables whole program compilation in the PathScale EKOPath compiler. The mechanism requires a different compilation model than separate compilation. This new mode of compilation is used when the `-ipa` flag is specified.

Whole program compilation requires the entire program to be presented to the compiler for analysis and optimization. This is possible only after a link step is applied. Ordinarily, the link step is applied to `.o` files, after all optimization and code generation have been performed. In the IPA compilation model, the link step is applied very early in the compilation process, before most optimization and code generation. In this scenario, the program code being linked are not in the object code format. Instead, they are in the form of the intermediate representation (IR) used during compilation and optimization. After the program has been linked at the IR level, inter-procedural analysis and optimization are applied to the whole program. Subsequently, compilation continues with the backend phases to generate the final object code.

The IPA compilation model (see Figure 7.1) has been implemented with ease-of-use as one of its main objectives. At the user level, it is sufficient to just add the `-ipa` flag to both the compile line and the link line. Thus, users can avoid having to re-structure their Makefiles to use IPA. In order to do this, we have to introduce a new kind of `.o` files that we call IPA `.o`'s. These are `.o` files in which the program code is in the form of IR, and are different from ordinary `.o` files that contain object code. IPA `.o` files are produced when a file is compiled with the flags `-ipa -c`. IPA `.o` files can only be linked by the IPA linker. The IPA linker is invoked by adding the `-ipa` flag to the link command. This appears as if it is the final link step. In reality, this link step performs the following tasks:

1. Invokes the IPA linker
2. Performs inter-procedural analysis and optimization on the linked program
3. Invokes the backend phases to optimize and generate the object code
4. Invokes the real linker to produce the final executable.

Under IPA compilation, the user will notice that the compilation of separate files proceeds very fast, because it does not involve the backend phases. On the other hand, the linking phase will appear much slower because it now encompasses the compilation and optimization of the entire program.

7.3.2 Inter-procedural analysis and optimization

We call the phase that operates on the IR of the linked program IPA, for Inter-Procedural Analysis, but its tasks can be divided into two categories:

- Analysis to collect information over the entire program
- Optimization to transform the program so it can run faster

7.3.2.1 Analysis

IPA first constructs the program call graph. Each node in the call graph corresponds to a function in the program. The call graph represents the caller-callee relationship in the program.

Once the call graph is built, based on different inlining heuristics, IPA prepares a list of function calls where it wants to inline the callee into the caller.

Based on the call graph, IPA computes the mod-ref information for the program variables. This represents the information as to whether a variable is modified or referenced inside a function call.

IPA also computes alias information for all the program variables. Whenever a variable has its address taken, it can potentially be pointed to by a pointer. Places that dereference or store through the pointer potentially access the variable. IPA's alias analysis keeps track of this information so that in the presence of pointer accesses, as few variables are affected as possible so they can be optimized more aggressively.

The mod-ref and alias information collected by IPA are not just used by IPA itself. The information is also recorded in the program representation so the optimizations in the backend phases also benefit.

7.3.3 Optimization

The most important optimization performed by IPA is *inlining*, in which the call to a function is replaced by the actual body of the function. Inlining is most versatile in IPA because all the user function definitions are visible to it. Apart from eliminating the function call overhead, inlining increases optimization opportunities of the backend phases by letting them work on larger pieces of code. For instance, inlining may result in the formation of a loop nest that enables aggressive loop transformations.

Inlining requires careful benefit analysis because overdoing it may result in performance degradation. The increased program size can cause higher instruction cache miss rate. If a function is already quite large, inlining may result in the compiler running out of registers, so it has to use memory more often, which causes program slow-down. In addition, too much inlining can slow down the later phases of the compilation process.

Many function calls pass constants (including addresses of variables) as parameters. Replacing a formal parameter by its known constant value helps in the optimization of the function body. Very often, part of the code of the function can be determined useless and deleted. *Function cloning* creates different clones of a function with its parameters customized to the forms of the calls. It provides a subset of the benefits of inlining without increasing the size of the function that contains the call. Like inlining, it also increases the total size of the program.

If IPA can determine that all the calls pass the same constant parameter, it will perform constant propagation for the parameter. This has the same benefit as

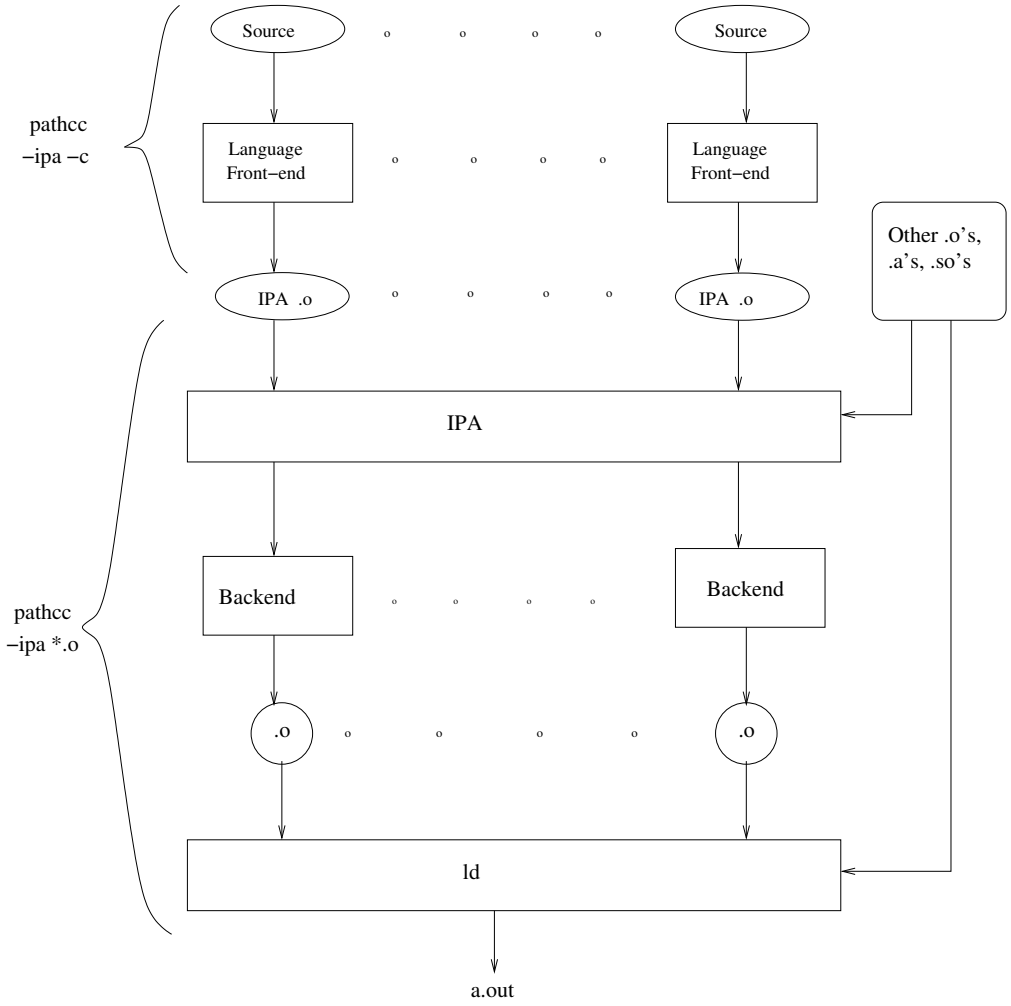

Figure 7.1: IPA Compilation Model

function cloning but does not increase the size of the program. Constant propagation also applies to global variables. If a global variable is found to be constant throughout the entire program execution, IPA will replace the variable by the constant value.

Dead variable elimination finds global variables that are never used over the program and deletes them. These variables are often exposed due to IPA's constant propagation.

Dead function elimination finds functions that are never called and deletes them. They can be the by-product of inlining and cloning.

Common padding applies to common blocks in Fortran programs. Ordinarily, compilers are incapable of changing the layout of the user variables in a common block, because this has to be co-ordinated among all the subroutines that use the same common block, and the subroutines may belong to different compilation units. But under IPA, all the subroutines are available. The padding improves the alignments of the arrays so they can be accessed more efficiently and even vectorized. The padding can also reduce data cache conflicts during execution.

Common block splitting also applies to common blocks in Fortran programs. This splits a common block into a number of smaller blocks which also reduces data cache conflicts during execution.

Procedure re-ordering lays out the functions of the program in an order based on their call relationship. This can reduce thrashing in the instruction cache during execution.

7.3.4 Controlling IPA

Although the compiler tries to make the best decisions regarding how to optimize a program, it is hard to make the optimal choice in general. Thus, the compiler provides many compilation options so the user can use them to tune for the peak performance of his program. This section presents the IPA-related compilation options that are useful in tuning programs.

But first, it is worthwhile to mention that IPA is one of the compilation phases that can benefit substantially from feedback compilation. In feedback compilation, a feedback data file containing a profile of a typical run of the program is presented to the compiler. This enables IPA to make better decisions regarding what functions to inline and clone. By ensuring that busy callers and callees are placed next to each other, IPA's procedure re-ordering can also be more effective. Feedback compilation is enabled by the `-fb-create` and `-fb-opt` options. See Section 7.6 for more details.

7.3.4.1 Inlining

There are actually two incarnations of the inliner in the PathScale EKOPath compiler, depending on whether `-ipa` is specified. This is because inlining is nowadays a language feature, and has to be performed independent of IPA. The inliner invoked when `-ipa` is not specified is the lightweight inliner, and it can only operate on a single compilation unit. The lightweight inliner does not do

automatic inlining. It inlines strictly according to the C++ language requirement, *C inline* keyword or any `-INLINE` options specified by the user. It may be invoked by default. The basic options to control inlining in the lightweight inliner are:

`-inline or -INLINE` causes the lightweight inliner to be invoked when `-ipa` is not specified.

`-INLINE:=off` suppresses the invocation of the lightweight inliner.

The options below are applicable to both the lightweight inliner and IPA's inliner:

`-INLINE:all` performs all possible inlining. Since this results in code bloat, this should only be used if the program is small.

`-INLINE:list=ON` makes the inliner list its actions on the fly. This is an useful option for the user to find out which functions are getting inlined, which functions are not being inlined and why. Thus, if the user wants to inline or not inline a function, tweaking the inlining controls based on the reasons specified by the output of this flag should help.

`-INLINE:must=`*`name1`*`[,`*`name2`*`,...]` forces inlining for the named functions.

`-INLINE:never=`*`name1`*`[,`*`name2`*`,...]` suppresses inlining for the named functions.

When `-ipa` is specified, IPA will invoke its own inliner and the lightweight inliner is not invoked. IPA's inliner automatically determines additional functions to inline in addition to those that are required. Small callees or callers are favored over larger ones. If profile data is available, calls executed more frequently are preferred. Otherwise, calls inside loops are preferred. Leaf routines (functions containing no call) are also favored. Inlining continues until no more call satisfies the inlining criteria, which can be controlled by the inlining options:

`-IPA:inline=OFF` turns off IPA's inliner, and the lightweight inliner is also suppressed since IPA is invoked. Default is `ON`.

`-INLINE:none` turns off automatic inlining by IPA but required inlining implied by the language or specified by the user are still performed. By default, automatic inlining is turned `ON`.

`-IPA:specfile=`*`filename`* directs the compiler to open the given file to read more `-IPA:` or `-INLINE:` options.

The following options can be used to tune the aggressiveness of the inliner. Very aggressive inlining can cause performance degradation as discussed in Section 7.3.3.

`-OPT:Olimit=N` specifies the size limit `N`, where `N` is computed from the number of basic blocks that make up a function; inlining will never cause a function to exceed this size limit. The default is 6000 under `-O2` and 9000 under `-O3`. The value `0` means no limit is imposed.

`-IPA:space=N` specifies that inlining should continue until a factor of `N%` increase in code size is reached. The default is `100%`. If the program size is small, the value of `N` could be increased.

`-IPA:plimit=N` suppresses inlining into a function once its size reaches `N`, where `N` is measured in terms of the number of basic blocks and the number of calls inside a function. The default is `2500`.

`-IPA:small_pu=N` specifies that a function with size smaller than `N` basic blocks is not subject to the `-IPA:plimit` restriction. The default is `30`.

`-IPA:callee_limit=n` specifies that a function whose size exceeds this limit will never be automatically inlined by IPA. The default is `500`.

`-IPA:min hotness=N` is applicable only under feedback compilation. A call site's invocation count must be at least `N` before it can be inlined by IPA. The default is `10`.

`-INLINE:aggressive=ON` increases the aggressiveness of the inlining, in which more non-leaf and out-of-loop calls are inlined. Default is `OFF`.

We mentioned that leaf functions are good candidates to be inlined. These functions do not contain calls that may inhibit various backend optimizations. To amplify the effect of leaf functions, IPA provides two options that exploit its call-tree-based inlining feature. This is based on the fact that a function that calls only leaf functions can become a leaf function if all of its calls are inlined. This in turn can be applied repeatedly up the call graph. In the description of the following two options, a function is said to be at *depth `N`* if it is never more than `N` edges from a leaf node in the call graph. A leaf function has depth `0`.

`-IPA:maxdepth=N` causes IPA to inline all routines at depth `N` in the call graph subject to space limitation.

`-IPA:forcedepth=N` causes IPA to inline all routines at depth `N` in the call graph regardless of space limitation.

7.3.5 Cloning

There are two options for controlling cloning:

`-IPA:multi_clone=N` specifies the maximum number of clones that can be created from a single function. The default is `0`, which implies that cloning is turned `OFF` by default.

`-IPA:node_bloat=N` specifies the maximum percentage growth in the number of procedures relative to the original program that cloning can produce. The default is `100`.

7.3.6 Other IPA tuning options

The following are options un-related to inlining and cloning, but useful in tuning:

`-IPA:common_pad_size=N` specifies that common block padding should use pad size of up to `N` bytes. The default value is `0`, which specifies that the compiler will determine the best padding size.

`-IPA:linear=ON` enables linearization of array references. When inlining Fortran subroutines, IPA tries to map formal array parameters to the shape of the actual parameters. The default is `OFF`, which means IPA will suppress the inlining if it cannot do the mapping. Turning this option `ON` instructs IPA to still perform the inlining but linearizes the array references. Such linearization may cause performance problems, but the inlining may produce more performance gain.

`-IPA:pu_reorder=N` controls IPA's procedure reordering optimization. A value of `0` disables the optimization. `N = 1` enables reordering based on the frequency in which different procedures are invoked. `N = 2` enables procedure reordering based on caller-callee relationship. The default is `0`.

`-IPA:field_reorder=ON` enables IPA's field reordering optimization to minimize data cache misses. This optimization is based on reference patterns of fields in large structs, learned during feedback compilation. The default is `OFF`.

`-IPA:ctype=ON` optimizes interfaces to constructs defined in the standard header file `ctype.h` by assuming that the program will not run in a multi-threaded environment. The default is `OFF`.

7.3.6.1 Disabling options

The following options are for disabling various optimizations in IPA. They are useful for studying the effects of the optimizations.

`-IPA:alias=OFF` disables IPA's alias and mod-ref analyses

`-IPA:addressing=OFF` disables IPA's address-taken analysis, which is a component of the alias analysis

`-IPA:cgi=OFF` disables the constant propagation for global variables (constant global identification)

`-IPA:cprop=OFF` disables the constant propagation for parameters

`-IPA:dfe=OFF` disables dead function elimination

`-IPA:dve=OFF` disables dead variable elimination

`-IPA:split=OFF` disables common block splitting

7.3.7 Case study on SPEC CPU2000

This section presents experimental data to show the importance of IPA in improving program performance. Our experiment is based on the SPEC CPU2000 benchmark suite compiled using release 1.2 of the Pathscale EKOPath compiler. The compiled benchmarks are run on a 1.4 GHz Opteron system. Two sets of data are shown here. The first set studies the effects of using the single option `-ipa`. The second set shows the effects of additional IPA-related tuning flags on the same files.

Table 7.1: Effects of IPA on SPEC CPU 2000 performance

Benchmark	Time w/o -ipa	Time with -ipa	Improvement %
164.gzip	170.7 s	164.7 s	3.5%
175.vpr	202.4 s	192.3 s	5%
176.gcc	113.6 s	113.2 s	0.4%
181.mcf	391.9 s	390.8 s	0.3%
186.crafty	83.5 s	83.4 s	0.1%
197.parser	301.4 s	289.3 s	4%
252.eon	152.8 s	126.8 s	17%
253.perlbmk	196.2 s	192.3 s	2%
254.gap	153.5 s	128.6 s	16.2%
255.vortex	175.2 s	132.1 s	24.6%
256.bzip2	210.2 s	181.0 s	13.9%
300.twolf	376.5 s	362.2 s	3.8%
168.wupwise	220.0 s	161.5 s	26.6%
171.swim	181.4 s	180.7 s	0.4%
172.mgrid	184.7 s	182.3 s	1.3%
173.applu	282.5 s	245.2 s	13.2%
177.mesa	155.4 s	131.5 s	15.4%
178.galgel	150.4 s	149.9 s	0.3%
179.art	245.7 s	221.1 s	10%
183.equake	143.7 s	143.2 s	0.3%
187.facerec	154.3 s	147.4 s	4.5%
188.ammp	266.5 s	261.7 s	1.8%
189.lucas	165.9 s	167.9 s	-1.2%
191.fma3d	239.6 s	244.6 s	-2.1%
200.sixtrack	265.0 s	276.9 s	-4.5%
301.apsi	280.7 s	273.7 s	2.5%

Table 7.1 shows how `-ipa` effects the base runs of the CPU2000 benchmarks. IPA improves the running times of 17 out of the 26 benchmarks; the improvements range from 1.3% to 26.6%. There are six benchmarks that improve by less than 0.5%, which is within the noise threshold. There are three FP benchmarks that slow down from 1.2% to 4.5% due to `-ipa`. The slowdown indicates that the benchmarks do not benefit from the default settings of the IPA parameters. By using additional IPA tuning flags, such slowdown can often be converted to performance gain. The average performance improvement over all the benchmarks listed in Table 7.1 is 6%.

Table 7.2: Effects of IPA tuning on some SPEC CPU2000 benchmarks

Benchmark	Time: Peak flags w/o IPA tuning	Time: Peak flags with IPA tuning	Improvement%	IPA Tuning Flags
181.mcf	325.3 s	275.5 s	15.3%	-IPA:field_reorder=on
197.parser	296.5 s	245.2 s	17.3%	-IPA:ctype=on
253.perlbmk	195.1 s	177.7 s	8.9%	-IPA:min_hotness=5:plimit=20000
168.wupwise	147.7 s	129.7 s	12.2%	-IPA:space=1000:linear=on -IPA:plimit=50000:callee_limit=5000 -INLINE:aggressive=on
187.facerec	144.6 s	141.6 s	2.1%	-IPA:plimit=1800

Table 7.2 shows the effects of using additional IPA tuning flags on the peak runs of the CPU2000 performance. In the peak runs, each benchmark can be built with its own combination of any number of tuning flags. We started with the peak flags of the benchmarks used in PathScale's SPEC CPU2000 submission, and we found that five of the benchmarks are using IPA tuning flags. Table 7.1 lists these five benchmarks. The second column gives the running times if the IPA-related tuning flags are omitted. The third column gives the running times with the IPA-related tuning flags. The fifth column lists their IPA-related tuning flags. As this second table shows, proper IPA tuning can produce major improvements in applications.

7.3.8 Invoking IPA

Inter-procedural analysis is invoked in several possible ways: `-ipa`, `-IPA`, and implicitly via `-Ofast`. IPA can be used with any optimization level, but gives the biggest potential benefit when combined with `-O3`. The `-Ofast` flag turns on `-ipa` as part of its many optimizations.

When compiling with `-ipa` the `.o` files that are created are not regular `.o` files. IPA uses the `.o` files in its analysis of your program, and then does a second compilation using that information to optimize the executable.

The IPA linker checks to see if the entire program is compiled with the same set of optimization options. If different optimization options are used, IPA will give a warning:

```
Warning: Inconsistent optimization options detected between files involved in
```

For example, the following invocation will generate this warning for two C files `a.c` and `b.c`.

```
~ $ pathcc -O2 -ipa -c a.c
~ $ pathcc -O3 -ipa -c b.c
~ $ pathcc -ipa a.o b.o
```

The user can pass consistent optimization options to the individual compilations to remove the warning. In the above example, the user can either pass -O2 or pass -O3 to both the files.

The `-ipa` flag implies `-O2 -ipa` because `-O2` is the default. Flags like `-ipa` can be used in combination with a very large number of other flags, but some typical combinations with the `-O` flags are shown below:

`-O3 -ipa` or `-O2 -ipa` is a typical additional attempt at improved performance over the `-O3` or `-O2` flag alone. `-ipa` needs to be used both in the compile and in the link steps of a build.

Using IPA with your program is usually straightforward. If you have only a few source files, you can simply use it like this:

```
pathf95 -O3 -ipa main.f subs1.f subs2.f
```

If you compile files separately, the `*.o` files generated by the compiler do not actually contain object code; they contain a representation of the source code. Actual compilation happens at link time. The link command also needs the `-ipa` flag added.

For example, you could separately compile and then link a series of files like this:

```
pathf95 -c -O3 -ipa main.f
pathf95 -c -O3 -ipa subs1.f
pathf95 -c -O3 -ipa subs2.f
pathf95 -O3 -ipa main.o subs1.o subs2.o
```

Currently, there is a restriction that each archive (for example `libfoo.a`) must contain *either* `.o` files compiled with `-ipa` *or* `.o` files compiled without `-ipa`, but not both.

Note that, in a non-IPA compile, most of the time is incurred with compiling all the files to create the object files (the `.o`'s) and the link step is quite fast. In an IPA compile, the creating of `.o` files is very fast, but the link step can take a long time. The total compile time can be considerably longer with IPA than without.

When invoking the final link phase with `-ipa` (for example, `pathcc -ipa -o foo *.o`), significant portions of this process can be done in parallel on a system with multiple processing units. To use this feature of the compiler, use the `-IPA:max_jobs` flag.

Here are the options for the `-IPA:max_jobs` flag:

`-IPA:max_jobs=N` This option limits the maximum parallelism when invoking the compiler after IPA to (at most) `N` compilations running at once. The option can take the following values:

`0` = The parallelism chosen is equal to either the number of CPUs, the number of cores, or the number of hyperthreading units in the compiling system, whichever is greatest.

`1` = Disable parallelization during compilation (default)

`>1` = Specifically set the degree of parallelism

7.3.9 Size and correctness limitations to IPA

IPA often works well on programs up to 100,000 lines, but is not recommended for use in larger programs in this release.

7.4 Loop Nest Optimization (LNO)

If your program has many nests of loops, you may want to try some of the Loop Nest Optimization group of flags. This group defines transformations and options that can be applied to loop nests.

One of the nice features of the PathScale EKOPath compilers is that its powerful Loop Nest Optimization feature is invoked by default at `-O3`. This feature can provide up to a 10-20x performance advantage over other compilers on certain matrix operations at `-O3`.

In rare circumstances, this feature can make things slower, so you can use `-LNO:opt=0` to disable nearly all loop nest optimization. Trying to make an `-O2` compile faster by adding `-LNO:opt=on` will not work because the `-LNO` feature is only active with `-O3` (or `-Ofast` which implies `-O3`).

Some of the features that one can control with the `-LNO:` group are:

- Loop fusion and fission
- Blocking to optimize cache line reuse
- Cache management
- TLB (Translation Lookaside Buffer) optimizations
- Prefetch

In this section we will highlight a few of the LNO options that have frequently been valuable.

7.4.1 Loop fusion and fission

Sometimes loop nests have too few instructions and consecutive loops should be combined to improve utilization of CPU resources. Another name for this process is loop fusion.

Sometimes a loop nest will have too many instructions, or deal with too many data items in its inner loop, leading to too much pressure on the registers, resulting in spills of registers to memory. In this case, splitting loops can be beneficial. Like splitting an atom, splitting loops is termed fission. These are the LNO options to control these transformations:

`-LNO:fusion=n` Perform loop fusion, `n:` `0` off, `1` conservative, `2` aggressive. Level 2 implies that outer loops in consecutive loop nests should be fused, even if it is found that not all levels of the loop nests can be fused. The default level is `1` (standard outer loop fusion), but `2` has been known to benefit a number of well-known codes.

`-LNO:fission=n` Perform loop fission, `n:` `0` off, `1` standard, `2` try fission before fusion. The default level is `0`, but `2` has been known to benefit a number of well-known codes.

Be careful with mixing the above two flags, because fusion has some precedence over fission: if `-LNO:fission=[1 or 2]` and `-LNO:fusion=[1 or 2]` then fusion is performed.

`-LNO:fusion_peeling_limit=n` controls the limit for the number of iterations allowed to be peeled in fusion, where `n` has a default of 5 but can be any non-negative integer.

Peeling is done when the iteration counts in consecutive loops is different, but close, and several iterations are replicated outside the loop body to make the loop counts the same.

7.4.2 Cache size specification

The PathScale EKOPath compilers are primarily targeted at the Opteron CPU currently, so they assume an L2 cache size of 1MB. Athlon 64 can have either a 512KB or 1MB L2 cache size. If your target machine is Athlon 64 and you have the smaller cache size, then setting `-LNO:cs2=512k` could help. You can also specify your target machine instead, using `-march=athlon64`. That would automatically set the standard machine cache sizes.

Here is the more general description of some of what is available.

`-LNO:cs1=n`, `cs2=n`, `cs3=n`, `cs4=n`
This option specifies the cache size. `n` can be `0` or a positive integer followed by one of the following letters: `k`, `K`, `m`, or `M`. These letters specify the cache size in Kbytes or Mbytes.

Specifying `0` indicates there is no cache at that level.
`cs1` is the primary cache
`cs2` refers to the secondary cache
`cs3` refers to memory
`cs4` is the disk

Default cache size for each type of cache depends on your system. Use `-LIST:options=ON` to see the default cache sizes used during compilation.

With a smaller cache, the cache set associativity is often decreased as well. The flag set: `-LNO:assoc1=n`, `assoc2=n`, `assoc3=n`, `assoc4=n` can define this appropriately for your system.

Once again, the above flags are already set appropriately for Opteron.

7.4.3 Cache blocking, loop unrolling, interchange transformations

Cache blocking, also called 'tiling', is the process of choosing the appropriate loop interchanges and loop unrolling sizes at the correct levels of the loop nests so that cache reuse can be optimized and memory accesses reduced. This whole LNO feature is on by default, but can be turned off with: `-LNO:blocking=off`. `-LNO:blocking_size=n` specifies a block size that the compiler must use when

performing any blocking, where n is a positive integer that represents the number of iterations.

-LNO:interchange is on by default, but setting this =0 can disable the loop interchange transformation in the loop nest optimizer.

The LNO group controls outer loop unrolling, but the -OPT: group controls inner loop unrolling. Here are the major -LNO: flags to control loop unrolling:

-LNO:outer_unroll_max,ou_max=n specifies that the compiler may unroll outer loops in a loop nest by up to n per loop, but no more. The default is 10.

-LNO:ou_prod_max=n
Indicates that the product of unrolling levels of the outer loops in a given loop nest is not to exceed n, where n is a positive integer. The default is 16.

To be more specific about how much unrolling is to be done, use -LNO:outer_unroll,ou=n. This indicates that exactly n outer loop iterations should be unrolled, if unrolling is legal. For loops where outer unrolling would cause problems, unrolling is not performed.

7.4.4 Prefetch

The LNO group can provide guidance to the compiler about the level and type of prefetching to enable. General guidance on how aggressively to prefetch is specified by -LNO:prefetch=n, where n=1 is the default level. n=0 disables prefetching in loop nests, while n=2 means to prefetch more aggressively than the default.

-LNO:prefetch_ahead=n defines how many cache lines ahead of the current data being loaded should be prefetched. The default is n=2 cache lines.

7.4.5 Vectorization

Vectorization is an optimization technique that works on multiple pieces of data at once. For example, the compiler will turn a loop computing the mathematical function sin() into a call to the vsin() function, which is twice as fast.

The use of vectorized versions of functions in the math library like sin(), cosin() is controlled by the flag -LNO:vintr=0|1|2. 0 will turn off vectorization of math intrinsics, while 1 is the default. Under -LNO:vintr=2 the compiler will vectorize all math functions. Note that vintr=2 could be unsafe in that the vector forms of some of the functions could have accuracy problems.

Vectorization of user code (excluding these mathematical functions) is controlled by the flag -LNO:simd[=(0|1|2)], which enables or disables inner loop vectorization. 0 turns off the vectorizer, 1 (the default) causes the compiler to vectorize only if it can determine that there is no undesirable performance impact due to sub-optimal alignment, and 2 will vectorize without any constraints (this is the most aggressive).

-LNO:simd_verbose=ON prints vectorizer information (from vectorizing user code) to stdout. -LNO:vintr_verbose=ON prints information about whether or not the math intrinsic functions were vectorized.

See the eko man page for more information.

7.5 Code Generation (-CG:)

The code generation group governs some aspects of instruction-level code generation that can have benefits for code tuning.

`-CG:gcm=OFF` turns off the instruction-level global code motion optimization phase. The default is `ON`.

`-CG:load_exe=n` specifies the threshold for subsuming a memory load operation into the operand of an arithmetic instruction. The value of `0` turns off this subsumption optimization. By default this subsumption is performed only when the result of the load has only one (`n=1`) use. This subsumption is not performed if the number of times the result of the load is used exceeds the value `n`, a non-negative integer. We have found that `load_exe=2` or `0` are occasionally profitable. The default for 64-bit ABI and Fortran is `n=2`; otherwise the default is `n=1`.

`-CG:use_prefetchnta=ON` means for the compiler to use the prefetch operation that assumes that data is Non-Temporal at All (NTA) levels of the cache hierarchy. This is for data streaming situations in which the data will not need to be re-used soon. Default is `OFF`.

7.6 Feedback Directed Optimization (FDO)

Feedback directed optimization uses a special instrumented executable to collect profile information about the program; for example, it records how frequently every `if()` statement is true. This information is then used in later compilations to tune the executable.

FDO is most useful if a program's typical execution is roughly similar to the execution of the instrumented program on its input data set; if different input data has dramatically different `if()` frequencies, using FDO might actually slow down the program. This section also discusses how to invoke this feature with the `-fb-create` and `-fb-opt` flags.

NOTE: If the `-fb-create` and `-fb-opt` compiles are done with different compilation flags, it may or may not work, depending on whether the different compilation flags cause different code to be seen by the phase that is performing the instrumentation/feedback. We recommend using the same flags for both instrumentation and feedback.

FDO requires compiling the program at least twice. In the first pass:

```
pathcc -O3 -ipa -fb-create fbdata -o foo foo.c
```

The executable `foo` will contain extra instrumentation library calls to collect feedback information; this means `foo` will actually run a bit slower than normal. We are using `fbdata` for the file name in this example; you can use any name for your file.

Next, run the program `foo` with an example dataset:

```
./foo <typical_input_data>
```

During this run, a file with the prefix "fbdata" will be created, containing feedback information. The file name you use will become the prefix for your output file. For example, the output file from this example dataset might be named fbdata.instr0.ab342. Each file will have a unique string as part of its name so that files can't be overwritten.

To use this data in a subsequent compile:

```
pathcc -O3 -ipa -fb-opt fbdata -o foo foo.c
```

This new executable should run faster than a non-FDO foo, and will not contain any instrumentation library calls.

Experiment to see if FDO provides significant benefit for your application.

More details on feedback compilation with the PathScale EKOPath compilers can be found under the -fb-create and -fb-opt options in the eko man page.

7.7 Aggressive optimizations

The PathScale EKOPath Compiler Suite, like all modern compilers, has a range of optimizations. Some produce identical program output to the original, some can change the program's behavior slightly. The first class of optimizations is termed "safe" and the second "unsafe". As a general rule, our -O1,-O2,-O3 flags only perform "safe" optimizations. But the use of "unsafe" optimizations often can produce a good speedup in a program, while producing a sufficiently accurate result.

Some "unsafe" optimizations may be "safe" depending on the coding practices used. We recommend first trying "safe" flags with your program, and then moving on to "unsafe" flags, checking for incorrect results and noting the benefit of unsafe optimizations.

Examples of unsafe optimizations include the following.

7.7.1 Alias analysis

Both C and Fortran have occasions where it is possible that two variables might occupy the same memory. For example, in C, two pointers might point to the same location, such that writing through one pointer changes the value of the variable pointed to by another. While the C standard prohibits some kinds of aliasing, many real programs violate these rules, so the aliasing behavior of PathScale's compiler is controlled by the -OPT:alias flag. See Section 7.7.4.2 for more information.

Aliases are hidden definitions and uses of data due to:

- Accesses through pointers
- Partial overlap in storage locations (e.g. unions in C)
- Procedure calls for non-local objects
- Raising of exceptions

The compiler normally has to assume that aliasing will occur. The compiler does alias analysis to identify when there is no alias, so later optimizations can be performed. Certain C and C++ language rules allow some levels of alias analysis. Fortran has additional rules which make it possible to rule out aliasing in more situations: subroutine parameters have no alias, and side effects of calls are limited to global variables and actual parameters.

For C or C++, the coding style can help the compiler make the right assumptions. Using type qualifiers such as `const`, `restrict`, or `volatile` can help the compiler. Furthermore, if you supply some assumptions to make concerning your program, more optimizations can then be applied. The following are some of the various aliasing models you can specify, listed in order of increasingly stringent, and potentially dangerous, assumptions you are telling the compiler to make about your program:

`-OPT:alias=any` the default level, which implies that any two memory references can be aliased.

`-OPT:alias=typed` means to activate the ANSI rule that objects are not aliased it they have different base types. This option is activated by `-Ofast`.

`-OPT:alias=unnamed` assumes that pointers never to point to named objects.

`-OPT:alias=restrict` tells the compiler to assume that all pointers are restricted pointers and point to distinct non-overlapping objects. This allows the compiler to invoke as many optimizations as if the program were written in Fortran. A restricted pointer behaves as though the C '`restrict`' keyword had been used with it in the source code.

`-OPT:alias=disjoint` says that any two pointer *expressions* are assumed to point to distinct, non-overlapping objects.

To make the opposite assertion about your program's behavior, put '`no_`' before the value. For example, `-OPT:alias=no_restrict` means that distinct pointers may point to overlapping storage.

Additional `-OPT:alias` values are relevant to Fortran programmers in some situations:

`-OPT:alias=cray_pointer` asserts that an object pointed to by a Cray pointer is never overlaid on another variable's storage. This flag also specifies that the compiler can assume that the pointed-to object is stored in memory before a call to an external procedure and is read out of memory at its next reference. It is also stored before a `END` or `RETURN` statement of a subprogram.

`-OPT:alias=parm` promises that Fortran parameters do not alias to any other variable. This is the default. `no_parm` asserts that parameter aliasing is present in the program.

7.7.2 Numerically unsafe optimizations

Rearranging mathematical expressions and changing the order or number of floating point operations can slightly change the result. Example:

```
A = 2. * X
B = 4. * Y
C = 2. * (X + 2. * Y)
```

A clever compiler will notice that `C = A + B`. But the order of operations is different, and so a slightly different `C` will be the result. This particular transformation is controlled by the `-OPT:roundoff` flag, but there are several other numerically unsafe flags.

Some options that fall into this category are:

The options that control IEEE behavior such as `-OPT:roundoff=N` and `-OPT:IEEE_arithmetic=N`. Here are a couple of others:

`-OPT:div_split=(ON|OFF)` This option enables or disables transforming expressions of the form `X/Y` into `X*(1/Y)`. The reciprocal is inherently less accurate than a straight division, but may be faster.

`-OPT:recip=(ON|OFF)` This option allows expressions of the form `1/X` to be converted to use the reciprocal instruction of the computer. This is inherently less accurate than a division, but will be faster.

These options can have performance impacts. For more information, see the `eko` manual page. You can view the manual page by typing `man eko` at the command line.

7.7.3 Fast-math functions

When `-OPT:fast_math=on` is specified, the compiler uses fast versions of math functions tuned for the processor. The affected math functions include `log`, `exp`, `sin`, `cos`, `sincos`, `expf`, and `pow`. In general, the accuracy is within 1 ulp of the fully precise result, though the accuracy may be worse than this in some cases. The routines may not raise IEEE exception flags. They call no error handlers, and denormal number inputs/outputs are typically treated as 0, but may also produce unexpected results. `-OPT:fast_math=on` is effected when `-OPT:roundoff` is set to 2 or above,

A different flag `-ffast-math` improves FP speed by relaxing ANSI & IEEE rules. `-fno-fast-math` tells the compiler to conform to ANSI and IEEE math rules at the expense of speed. `-ffast-math` implies `-OPT:IEEE_arithmetic=2 -fno-math-errno`, while `-fno-fast-math` implies `-OPT:IEEE_arithmetic=1 -fmath-errno`. These flags apply to all languages.

Both `-OPT:fast_math=on` and `-ffast-math` are implied by `-Ofast`.

7.7.4 IEEE 754 compliance

It is possible to control the level of IEEE 754 compliance through options. Relaxing the level of compliance allows the compiler greater latitude to transform the code for improved performance. The following subsections discuss some of those options.

7.7.4.1 Arithmetic

Sometimes it is possible to allow the compiler to use operations that deviate from the IEEE 754 standard to obtain significantly improved performance, while still obtaining results that satisfy the accuracy requirements of your application.

The flag regulating the level of conformance to ANSI/IEEE 754-1985 floating pointing roundoff and overflow behavior is:

`-OPT:IEEE_arithmetic=N` (where `N= 1`, `2`, or `3`).

`-OPT:IEEE_arithmetic`

`=1` Requires strict conformance to the standard

`=2` Allows use of any operations as long as exact results are produced. This allows less accurate inexact results. For example, `X*0` may be replaced by `0`, and `X/X` may replaced by `1` even though this is inaccurate when `X` is `+inf`, `-inf`, or `NaN`. This is the default level at `-O3`.

`=3` Means to allow any mathematically valid transformations. For example, replacing `x/y` by `x*(recip(y))`.

For more information on the defaults for IEEE arithmetic at different levels of optimization, see Table 7.3.

7.7.4.2 Roundoff

Use `-OPT:roundoff=` to identify the extent of roundoff error the compiler is allowed to introduce:

`0` No roundoff error

`1` Limited roundoff error allowed

`2` Allow roundoff error caused by re-associating expressions

`3` Any roundoff error allowed

The default roundoff level with `-O0`, `-O1`, and `-O2` is `0`. The default roundoff level with `-O3` is `1`.

Listing some of the other `-OPT:` sub-options that are activated by various roundoff levels can give more understanding about what the levels mean.

`-OPT:roundoff=1` implies:

- `-OPT:fast_exp=ON` This option enables optimization of exponentiation by replacing the run-time call for exponentiation by multiplication and/or square root operations for certain compile-time constant exponents (integers and halves).
- `-OPT:fast_trunc` implies inlining of the `NINT`, `ANINT`, `AINT`, and `AMOD` Fortran intrinsics.

`-OPT:roundoff=2` turns on the following sub-options:

- `-OPT:fold_reassociate` which allows optimizations involving re-association of floating-point quantities.

`-OPT:roundoff=3` turns on the following sub-options:

- `-OPT:fast_complex` When this is set `ON`, complex absolute value (norm) and complex division use fast algorithms that overflow for an operand (the divisor, in the case of division) that has an absolute value that is larger than the square root of the largest representable floating-point number.
- `-OPT:fast_nint` uses a hardware feature to implement single and double-precision versions of `NINT` and `ANINT`

7.7.5 Other unsafe optimizations

A few advanced optimizations intended to exploit some exotic instructions such as `CMOVE` (conditional move) result in slightly changed program behavior, such as programs which write into variables guarded by an `if()` statement. For example:

```
if (a .eq. 1) then
   a = 3
endif
```

In this example, the fastest code on an x86 CPU is code which avoids a branch by always writing `a`; if the condition is false, it writes `a`'s existing value into `a`, else it writes `3` into `a`. If a is a read-only value not equal to `1`, this optimization will cause a segmentation fault in an odd but perfectly valid program.

7.7.6 Assumptions about numerical accuracy

See the following table for the assumptions made about numerical accuracy at different levels of optimization.

Table 7.3: **Numerical accuracy with options**

-OPT: option name	**-O0**	**-O1**	**-O2**	**-O3**	**-Ofast**	Notes
div_split	off	off	off	off	on	on if IEEE_a=3
fast_complex	off	off	off	off	off	on if roundoff=3
fast_exp	off	off	off	on	on	on if roundoff>=1
fast_nint	off	off	off	off	off	on if roundoff=3
fast_sqrt	off	off	off	off	off	
fast_trunc	off	off	off	on	on	on if roundoff>=1
fold_reassociate	off	off	off	off	on	on if roundoff>=2
fold_unsafe_relops	on	on	on	on	on	
fold_unsigned_relops	off	off	off	off	off	
IEEE_arithmetic	1	1	1	2	2	
IEEE_NaN_inf	off	off	off	off	off	
recip	off	off	off	off	on	on if roundoff>=2
roundoff	0	0	0	1	2	
fast_math	off	off	off	off	off	on if roundoff>=2
rsqrt	0	0	0	0	1	1 if roundoff>=2

For example, if you use -OPT:IEEE_arithmetic at -O3, the flag is set to IEEE_arithmetic=2 by default.

7.8 Hardware performance

Although the x86_64 platform has excellent performance, there are a number of subtleties in configuring your hardware and software that can each cause substantial performance degradations. Many of these are not obvious, but they can reduce performance by 30% or more at a time. We have collected a set of techniques for obtaining best performance described below.

7.8.1 Hardware setup

There is no "catch all" memory configuration that works best across all systems. We have seen instances where the number, type, and placement of memory modules on a motherboard can each affect the memory latency and bandwidth that you can achieve.

Most motherboard manuals have tables that document the effects of memory placement in different slots. We recommend that you read the table for your motherboard, and experiment.

If you fail to set up your memory correctly, this can account for up to a factor-of-two difference in memory performance. In extreme cases, this can even affect system stability.

7.8.2 BIOS setup

Some BIOSes allow you to change your motherboard's memory interleaving options. Depending on your configuration, this may have an effect on performance.

For a discussion of memory interleaving across nodes, see Section 7.8.3 below.

7.8.3 Multiprocessor memory

Traditional small multiprocessor (MP) systems use symmetric multiprocessing (SMP), in which the latency and bandwidth of memory is the same for all CPUs.

This is not the case on Opteron multiprocessor systems, which provide non-uniform memory access, known as NUMA. On Opteron MP systems, each CPU has its own direct-attached memory. Although every CPU can access the memory of all others, memory that is physically closest has both the lowest latency and highest bandwidth. The larger the number of CPUs, the higher will be the latency and the lower the bandwidth between the two CPUs that are physically furthest apart.

Most multiprocessor BIOSes allow you to turn on or off the interleaving of memory across nodes. Memory interleaving across nodes masks the NUMA variation in behavior, but it imposes uniformly lower performance. We recommend that you turn node interleaving off.

7.8.4 Kernel and system effects

To achieve best performance on a NUMA system, a process or thread and as much as possible of the memory that it uses must be allocated to the same single CPU. The Linux kernel has historically had no support for setting the affinity of a process in this way.

Running a non-NUMA kernel on a NUMA system can result in changes in performance while a program is running, and non-reproducibility of performance across runs. This occurs because the kernel will schedule a process to run on whatever CPU is free without regard to where the process's memory is allocated.

Recent kernels have some degree of NUMA support. They will attempt to allocate memory local to the CPU where a process is running, but they still may not prevent that process from later being run on a different CPU after it has allocated memory. Current NUMA-aware kernels do not migrate memory across NUMA nodes, so if a process moves relative to its memory, its performance will suffer in unpredictable ways.

Note that not all vendors ship NUMA-aware kernels or C libraries that can interface to them. If you are unsure of whether your kernel supports NUMA, check with your distribution vendor.

7.8.5 Tools and APIs

Recent Linux distributions include tools and APIs that allow you to bind a thread or process to run on a specific CPU. This provides an effective workaround for the problem of the kernel moving a process away from its memory.

Your Linux distribution may come with a package called `schedutils`, which includes a program called `taskset`. You can use `taskset` to specify that a program must run on one particular CPU.

For low-level programming, this facility is provided by the `sched_setaffinity(2)` call in the C library. You will need a recent C library to be able to use this call.

On systems that lack NUMA support in the kernel, and on runs that do not set process affinity before they start, we have seen variations in performance of 30% or more between individual runs.

7.8.6 Testing memory latency and bandwidth

To test your memory latency and bandwidth, we recommend two tools.

For memory latency, the LMbench package provides a tool called `lat_mem_rd`. This provides a cryptic, but fairly accurate, view of your memory hierarchy latency.

LMbench is available from `http://www.bitmover.com/lmbench/`

For measuring memory bandwidth, the `STREAM` benchmark is a useful tool. Compiling either the Fortran or C version of the benchmark with the following command lines will provide excellent performance:

```
$ pathf95 -Ofast stream_d.f second_wall.c -DUNDERSCORE
$ pathcc -Ofast -lm stream_d.c second_wall.c
```

(If you do not compile with at least `-O3`, performance may drop by 40% or more.)

The `STREAM` benchmark is available from `http://www.streambench.org/`

For both of these tools, we recommend that you perform a number of identical runs and average your results, as we have observed variations of more than 10% between runs.

7.9 The `pathopt2` tool

The `pathopt2` tool is used to iteratively test different options and option combinations by compiling a set of application source code files, measuring the performance of the executable and tracking the results. The best options are obtained from the output of these runs and are used to adaptively tune successive runs, yielding the best set of compiler options for a given combination of application code, data set, hardware, and environment. A sorted list of execution times is produced for each run.

The tool uses an XML option configuration file that defines one or more execution targets. Each execution target specifies options to try and indicates how they are to be combined into a series of tests. In general, using `pathopt2` involves these steps:

1. Run pathopt2 using an execution target in the supplied option configuration file.
2. Interpret the results.
3. Choose a more detailed execution target based on the results from the first run, and repeat the process until the best compiler options are found.

The pathopt2 tool can be completely driven from its command line, or it can alternatively use scripts to build and test the programs. Scripts are useful for more complex runs, for interfacing to existing build and test mechanisms, and for automating the process. For a standard installation, the program pathopt2 is located in:

```
/opt/pathscale/bin
```

This is the same directory that contains pathcc, pathCC, pathf95, pathf90, and so on.

An option configuration file, pathopt2.xml, is provided. The default location is:

```
/opt/pathscale/share/pathopt2/pathopt2.xml
```

See Section 7.9.3 for details on this file format. Sample programs are found in:

```
/opt/pathscale/share/pathopt2/examples
```

In the following sections we review the command syntax, the option configuration file structure, and general usage information. Step-by-step examples show how to use the different features of pathopt2.

7.9.1 A simple example

An example is provided here to show basic usage of pathopt2. In this example you will copy a test program into your working directory, and then run pathopt2 with the options file and the test program.

Copy the program factorial.c from /opt/pathscale/share/pathopt2/examples into your own working directory. factorial.c is a program that calculates a table of 50,000 factorials, from 1! to 50000! You can now run this simple example by typing:

```
$  pathopt2 -f pathopt2.xml -t try5  \
     -r ./factorial pathcc @ -o factorial factorial.c
```

Note: If you do not have '.' set in your PATH, you need to use '`./factorial`' to run this command from the current working directory. The PATH for the program `pathopt2` is the same as for `pathcc`, etc., and should already be set correctly. See the *PathScale EKOPath Compiler Suite Install Guide* for general information on setting your PATH.

You should see a list of output summarizing the result of all the runs. The first set of flags are listed in the order in which they were run. This is followed by a summary table which sorts the same output by time, from fastest to slowest. Sample output from this run is shown below:

```
Flags                Build   Test   Real   User   System
-------------------  -----   -----  -----  -----  ------
-O2                  PASS    PASS   2.83   2.82    0.00
-O3                  PASS    PASS   2.39   2.39    0.00
-O3 -ipa             PASS    PASS   2.40   2.40    0.01
-O3 -OPT:Ofast       PASS    PASS   2.37   2.38    0.00
-Ofast               PASS    PASS   2.38   2.38    0.00

Sorted summary from all runs:
Flags                Build   Test   Real   User   System
-------------------  -----   -----  -----  -----  ------
-O3 -OPT:Ofast       PASS    PASS   2.37   2.38    0.00
-Ofast               PASS    PASS   2.38   2.38    0.00
-O3                  PASS    PASS   2.39   2.39    0.00
-O3 -ipa             PASS    PASS   2.40   2.40    0.01
-O2                  PASS    PASS   2.83   2.82    0.00
```

From these results, we see that the best option from this run is `-O3 -OPT:Ofast`. The next sections will discuss details on usage, command line options, and the configuration file format.

7.9.2 `pathopt2` usage

Basic usage is as follows:

```
pathopt2 [-n num_iterations] [-f configfile] [-t execute_target]
  [-r test_command] [-S real|user|system] build_command @ [args] ...
```

The command line above shows the most commonly used options; for the complete list of options, see Table 7.4. The `pathopt2` tool runs `build_command` with the provided arguments and using additional options as specified in `configfile`. The build command can be an EKOPath invocation command (`pathcc, pathf95, pathCC`), a `make` command, or a script which eventually invokes the compiler, perhaps via a `make` command. The character `@` is replaced in the command with the list of options from the `configfile` being considered. The `configfile` is typically the provided `pathopt2.xml` file, although you can write your own. The `execute_target` parameter specifies the execution target from the `configfile`. The `test_command` parameter is the command to run the program and can be replaced with a script. The program is expected to return a status value of 0 to indicate success, or a non-zero status to indicate failure.

The `-S` option specifies the metric used for comparing performance:

- real: the elapsed real time (this is the default).
- user: the CPU time spent executing in user mode.
- system: the CPU time spent executing in system mode.
- timing-file: to use a file containing a timing value.
- rate-file: to use a file containing a rate value.

The chosen metric is used to guide the choices made by the `pathopt2` algorithms when selecting options for the best performance, and is used to sort the final output.

The interpretation of real, user and system time is the same as the time(1) command. `real` is equivalent to wall-clock time. An application may switch back and forth between user and kernel mode so these components are factored separately into `user` and `system` times. Since the O/S is typically time-slicing between many processes, the sum of `user` and `system` does not necessarily equal `real` since other processes could also have run. The default metric used when comparing the performance of one set of options with another is `real` time. All 3 times will be displayed in the output.

Additionally, `pathopt2` allows arbitrary performance metrics to be used to guide option selection using the timing-file and rate-file choices. When either of these options is used, `pathopt2` sets an environment variable called PSC_METRIC_FILE with the name of a temporary file before running the command. The run command is required to write the performance metric into this file before it terminates. The `pathopt2` tool then opens this file, reads a value from the file as a double-precision floating-point number, and deletes the temporary file. The only interpretation placed on these values is that smaller is better for timing, and that larger is better for rate. The actual units of the values do not matter as far as `pathopt2` is concerned since it just performs comparisons on the values.

Using the above usage as a guide, we can now summarize the simple command from the previous section:

```
$ pathopt2 -f pathopt2.xml -t try5 \
    -r ./factorial pathcc @ -o factorial factorial.c
```

This example directs `pathopt2` to use `pathopt2.xml` as the configuration file. The build command `pathcc @ -o factorial factorial.c` is used for the building phase where option “@” is iteratively replaced with the rules specified in the `try5` subset within the configuration file `pathopt2.xml`. The “@” character must be included somewhere in the build command since this is the mechanism by which the chosen optimization options are propagated to the build command. Finally, `./factorial` is used as the `test_command`.

For simple cases, the `-o` flag can be omitted, and the default executable output `a.out` can be used as the `test_command`:

```
$  pathopt2 -f pathopt2.xml -t try5 \
     -r ./a.out pathcc @ factorial.c
```

Note: The order of the options in the command line does not matter. However, the required `build_command` comes last since it may have an arbitrary number of options and arguments of its own. When the `-f` option is not specified `pathopt2` will use the file `pathopt2.xml` if it is present in the current working directory, otherwise it will use the default `pathopt2.xml` that ships with the EKOPath software.

The `pathopt2` available options are given in Table7.4. You can also type:

```
$  pathopt2 -h
```

on the command line to get usage information.

Table 7.4: Options for pathopt2

Option	Description	Default
-D	Do not redirect I/O to /dev/null This is useful for debugging problems with the compilation, the run, or the build and test scripts.	All I/O from the build and test commands will be sent to /dev/null under the assumption that the program will build and run cleanly.
-f *configfile*	The -f option is used to specify the filename of the pathopt2 XML configuration file.	If it is not specified the tool will first check for a file called pathopt2.xml in the current working directory and use it if present, otherwise the tool will use the file <install_path>/pathscale/share/pathopt2/pathopt2.xml
-g *external_configfile*	Loads in additional user-defined configfile(s). This allows a user to extend the pathopt2.xml file without having to modify it.	
-h	Show usage	
-j	Number of jobs	1
-k	Keep temporary directory (with -T)	Remove temporary directory
-M	Directory name	'pwd'
-n *num_iterations*	Number of iterations to run on each option	1
-r *test_command*	Test script	If this option is not specified then there is no test run, and the performance of the build command is used. This is useful when the program is built and run in one step, and the timing-file or rate-file mechanism is used to report the performance.
-S|real|user|system| *timing_file* |*rate_file*	Selects the performance metric for choosing options and for sorting the results	real
-t *execute_target*	Use execute_target, which corresponds to an <execute> tag found in *configfile*.	The first target in *configfile*
-T	Run script in temporary directory	Do not use a temporary directory
-v	Generate more verbose output	
-w *columns*	Number of columns to use in formatting output	40
-X	Don't print out a summary table	

7.9.3 Option configuration file

The PathScale EKOPath Compiler Suite includes `pathopt2.xml`, a pre-configured option configuration file found in `/opt/pathscale/share/pathopt2/` that contains about 200 test flags and options. This XML file specifies a tree of options to try. A small set of tags and attributes are used. The file supports many common combinations of options in a framework that enables `pathopt2` to adapt as it runs. `pathopt2.xml` can be used on its own, or as a framework for creating a custom configuration file. More than one configuration can be described in a single file.

A single configuration in `pathopt2.xml` consists of two parts:

- A list of options. This list is contained within a `<define>` tag. This list can also contain any number of `<option>`, `<choose>`, or `<append>` tags.
- An execute target. This is a set of rules that accesses the named options list via the `<source/>` tag. The execute target can use multiple `<source/>` tags in order to combine different lists of options. It can also contain `<option>` or `<append>` tags.

An execute target can be addressed on the command line using the `-t` option. By default, `pathopt2` runs only the first execute target in a configuration file. The following is a listing of the `try5_list` and the `try5` execute target in the default `pathopt2.xml` file. `try5` is typically the first target to use when testing options with `pathopt2`.

```
 <define name="try5_list">
     <option> -O2 </option>
     <option> -O3 </option>
    <choose k="1">
       <append>
       <option> -O3 </option>
       <choose k="1">
          <option> -ipa </option>
          <option> -OPT:Ofast </option>
       </choose>
       </append>
    </choose>
    <option> -Ofast </option>
</define>

<execute name="try5">
   <choose k="1">
   <source from="try5_list"/>
   </choose>
</execute>
```

The first two options, `-O2` and `-O3` are run in order. Next, the `-O3` option is appended to both `-ipa` and `-OPT:Ofast.` Finally, `-Ofast` is used. This ordering is shown in the first part of the `pathopt2` output when `try5` is the target:

Flags	Build	Test	Real	User	System
-O2	PASS	PASS	2.83	2.82	0.00
-O3	PASS	PASS	2.39	2.39	0.00
-O3 -ipa	PASS	PASS	2.40	2.40	0.01
-O3 -OPT:Ofast	PASS	PASS	2.37	2.38	0.00
-Ofast	PASS	PASS	2.38	2.38	0.00

The list of the tags for the option configuration file is given in the following table.

Table 7.5: Tags for option configuration file

Tag	**Description**
<config> ... </config>	Main body tag describing the configuration. All other tags and attributes must reside inside this tag.
<execute name="*name*"> ... </execute	Specifies an execute target, and must a contain at least one <source/> tag that references a previously defined <define> tag. May also contain <option> or <append> tags. Specify execute targets on the command line using -t *name*.
<option> ... </option>	Describes a single option. Surround the content for this option in space characters to ensure differentiation, e.g. <option> -Ofast </option> rather than <option>-Ofast</option>
<choose k="k" [hoist="true"]> ... </choose>	Choose the best option among those provided within this tag. The k="*k*" attribute specifies the number of choices to run iteratively. If k is given as a range separated by a colon, e.g. k="0:2" pathopt2 chooses among that number of options, inclusive, e.g. between 0 and 2 options. The optional hoist="true" attribute merges the lists returned by the children of the <execute> tag into the list for that tag. By default, <choose> picks combinations only from directly-related children.
<append> <option>...</option> ... </append>	The first option described within this tag is appended to the test stream for the remaining options. The following instructs pathopt2 to find the best option between "-O3 -ipa" and "-O3 -OPT:Ofast", but not any of these options singly: <append> <option> -O3 </option> <choose k="1"> <option> -ipa </option> <option>-OPT:Ofast</option> </choose> </append>
<define name="*name*"> ... </define>	Defines a block of options that can be later included using the <source from="*name*"/> tag. Note that this block can include any number of <option>, <choose>, or <append> tags.
<source from="*name*"/>	Includes a block of options previously defined with <define>.
<bestof k="*k*"> <context> ... </context> <option>...</option> <option>...</option> </bestof>	Choose the best option in the list, referenced by run time and chosen in the context of the option listed in the <context> tag. The k option is used as described for the <choose> tag. <context> specifies an option to use as a basis for testing, but not to propagate to outside tags.
<!-- comment -->	Standard XML comment tag, ignored by the parser.

NOTE: All tags other than `<source/>` require an end tag (e.g. `<append>` requires a corresponding `</append>`).

7.9.4 Testing methodology

Typically, the execute target `try5` in `pathopt2.xml` is used first with the `pathopt2` command. After the results of the run are available, you can look for the fastest result of the 5 options, and then run `pathopt2` again with a new execute target. The next set of refinements in the execute targets are the options with the "`peak_`" prefix. For example, if the best results were obtained with `-O2`, then the next target to try will be `peak_O2`. Here is a summary of the target usage:

Option in `try5` with best results	**Use this target for next run**
`-O2`	`peak_O2`
`-O3`	`peak_O3`
`-O3 -OPT:Ofast`	`peak_O3`
`-O3 -ipa`	`peak_O3`
`-Ofast`	`peak_Ofast`

This progressive refinement is shown in more detail in 7.9.8.3 and in 7.9.8.4.

7.9.5 Using an external configuration file to modify pathopt2.xml

It is possible to build hierarchies of lists and to construct new execution targets by combining existing ones. The way to do this without modifying `pathopt2.xml` is to create an external configuration file, then use the `-g` option in the `pathopt2` command line to load it in. The XML files are processed in order as if they were concatenated. The `-g` option can be repeated to load in more than one file. The `-t` option chooses the execution target as before. The rules for using the `-f` option remain the same. Here is an example of an external configuration file that extends the `try5_list` with a 6th possibility:

```
<config>
   <execute name="try6">
      <choose k="1">
      <source from="try5_list"/>
      <option> -O1 </option>
      </choose>
   </execute>
</config>
```

7.9.6 PSC_GENFLAGS environment variable

The pathopt2 tool arranges that the specified options are passed through as arguments to the build command using the expansion of the “@” character on the pathopt2 command line. Usually these options will then be explicitly passed to the compiler, either directly or via a Makefile variable such as CFLAGS or FFLAGS. Alternatively, the PathScale compilers will also process options from the PSC_GENFLAGS environment variable. This provides a way to implicitly pass the pathopt2 selected options to the compiler through existing scripts and Makefiles without their modification. Note that pathopt2 itself does not set the value of PSC_GENFLAGS but it can be easily achieved using a shell script as the build command and using the syntax:

```
export PSC_GENFLAGS="$*"
```

7.9.7 Using build and test scripts

The first example was run without build or test scripts. However, scripts provide added flexibility to pathopt2. Here are three common reasons for using a build script:

- You might need to cd to another directory before issuing the make command.
- There may be several directories you need to go to to complete the build.
- There may be no ’make clean’ target, so you need a ’rm *.o’ command before the make command.

There are several reasons for using a test script:

- pathopt2 can’t handle a complicated program run command with whitespace in it.
- You may need to cd to another directory before running the program.
- You want to take advantage of the -S rate-file or -S timing-file feature; that requires some grep and sed commands to isolate the number in the output to use as the performance metric of interest: e.g. a megaflops number in the rate-file case.

The next sections provide examples of a Makefile, build and test scripts and the rate and timing files.

7.9.8 The NAS Parallel Benchmark Suite

Next is a concrete example with measurable results. The NAS Parallel Benchmark (NPB) suite is commonly used for both serial and parallel benchmarking. It consists of a set of dissimilar pieces of applications illustrating the various numerical techniques used by NASA’s high performance applications. The benchmark comes with several data set sizes, with W being a "workstation" size (smallest), and A and B being two sizes appropriate to a cluster or supercomputer-size problem. Thes examples uses the Class A data set.

Several examples will be provided, showing usage in a step-by-step mannner. By following these steps, you will get a better idea of how `pathopt2` works.

7.9.8.1 Set up the workarea

The NAS Parallel Benchmark Suite (NPB) can be downloaded by going to:

```
http://www/nas/nasa.gov/Software/NPB
```

and following the links to the file. Download the file to a writable working directory. Then:

```
$  tar zxf NPB2.3.tar.gz
$  cd NPB2.3/NPB2.3-SER/config
$  cp /opt/pathscale/share/pathopt2/examples/make.def .
$  cd ..
```

7.9.8.2 Example 1-Run with Makefile

This shows the simplest use of the application with a Makefile. There are no optimization flags in the `make.def` file we supply. All optimization flags are sent from `pathopt2` to the compiler by propagating the value of "@" from the `pathopt2` command line to the `CFLAGS` and `FFLAGS` Makefile variables.

The command will now look like this:

```
$  pathopt2 -t try5 -r bin/ft.A \
     make clean ft CLASS=A FFLAGS="@"
```

Note that we omitted the `-f pathopt2.xml` option in this example. As mentioned previously, when this option is omitted, `pathopt2` will use the file `pathopt2.xml` if it is present in the current working directory, otherwise it will use the default `pathopt2.xml` that ship with the EKOPath software.

Output from the run should be similar to the following. Only the sorted summary is shown here:

```
Sorted summary from all runs:
Flags                Build  Test   Real   User   System
-------------------  -----  -----  -----  -----  ------
-O3 -OPT:Ofast        PASS   PASS  12.74  12.38   0.36
-O3 -ipa              PASS   PASS  12.77  12.31   0.45
-O3                   PASS   PASS  12.79  12.42   0.37
-Ofast                PASS   PASS  13.66  13.18   0.47
-O2                   PASS   PASS  14.50  14.12   0.38
```

7.9.8.3 Example 2-Use build/run scripts and a timing file

Next, let's assume that we want to do our `pathopt2` work in a sub-directory of NPB2.3-SER to avoid littering the top-level directory with scripts and, possibly, output files.

```
$  mkdir pathopt2
$  cd pathopt2
$  mkdir logs
```

`logs` is where we will keep a copy of the last run of the `ft.A` executable. Copy the two scripts, `psc_build` and `psc_test` from `/opt/pathscale/share/pathopt2/examples` into the `pathopt2` directory. The scripts are shown below:

For `psc_build`:

```
#!/bin/sh
cd ..
make clean
code=$1
size=$2
shift 2
make $code CLASS=$size FFLAGS="$*"
cd pathopt2
```

For `psc_test`:

```
#!/bin/sh
../bin/ft.A > logs/ft.A.txt
```

Make the files executable and then run `pathopt2`:

```
$  chmod +x psc_*
$  pathopt2 -t try5 -r ./psc_test ./psc_build ft A @
```

Note that the first argument to the `psc_build` script is the name of the code, the second argument is the problem size and all remaining arguments are the optimization options. This matches the code in the `psc_build` script that interprets the arguments.

The output will be similar to the following:

```
Sorted summary from all runs:
Flags                Build  Test   Real   User   System
--------------------  ----  ----   ----   ----   ------
-O3 -ipa              PASS   PASS  12.67  12.23   0.44
-Ofast                PASS   PASS  12.68  12.27   0.40
-O3 -OPT:Ofast        PASS   PASS  12.83  12.39   0.44
-O3                   PASS   PASS  13.86  12.46   0.40
-O2                   PASS   PASS  14.53  14.14   0.39
```

It is useful to check the output in logs/ft.A.txt:

```
FT Benchmark completed:
Class                =                A
Size                 =     256x256x128
Iterations           =                6
Time in seconds      =            10.78
Mop/s total          =           662.05
Operation type       =  floating point
Verification         =      SUCCESSFUL
Version              =              2.3
```

Since -Ofast runs last in the try5 target, the output in this file corresponds to the 12.68 real or 12.27 user times from the -Ofast run. The reason the "Time in seconds" output by NPB is considerably lower than 12.68 is that it measures the time for the main work section of the program, ignoring the start-up and array initialization time. For the parallel versions of NPB, it is appropriate to ignore the initialization since that time does not improve when more processes are used in the computation.

This "Time in seconds" and "Mop/s total" (millions of operations per second) from the NPB benchmarks turn out to be useful metrics for testing optimization. The -S timing-file and rate-file features can be used to search for the "Time in seconds" or the "Mop/s total" metrics. In this next example we will use the timing-file option. See Example 3 in 7.9.8.4 for information on the rate-file option.

This "Time in seconds" output can be used as pathopt2's sorting criterion, by using the -S timing-file option. However, the psc_build script has to be enhanced to be able to isolate the number after the "Time in seconds =" part of the output. Here is how to do this in a script (found in /opt/pathscale/share/pathopt2/examples) called psc_test2:

```
#!/bin/sh
../bin/ft.A > logs/ft.A.txt
grep "in sec" logs/ft.A.txt > secs.log
sed -e 's/Time in seconds = //' secs.log > $PSC_METRIC_FILE
grep SUCCESSFUL logs/ft.A.txt
```

Note: pathopt2 checks the result status of the build command/script and of the run command/script. A zero status indicates that the build or run was successful, while a non-zero status indicates failure. If running the program indicates its status in some other way, this must be detected by a script and reflected in the script's return status. In the example above, the grep SUCCESSFUL line is a way to pass the NPB correctness test results to pathopt2. The grep will have a status of 0 if the output contains this phrase, and this will be the status of the whole shell script since this is the last command.

Next, make the file executable and run pathopt2:

```
$  chmod +x psc_test2
$  pathopt2 -S timing-file -t try5 -r ./psc_test2 \
     ./psc_build ft A @
```

The sorted summary will be similar to the following:

```
Sorted summary from all runs:
Flags                 Build   Test   Time
————————————————-     ————-   ————-   ————
-O3 -ipa              PASS    PASS   10.87
-Ofast                PASS    PASS   10.87
-O3 -OPT:Ofast        PASS    PASS   11.01
-O3                   PASS    PASS   11.02
-O2                   PASS    PASS   11.82
```

Since `-O3 -ipa` was the fastest in the `try5` target, we can run `pathopt2` again with the `peak_O3` target:

```
$  pathopt2 -S timing-file -t peak_O3 -r ./psc_test2 \
     ./psc_build ft A @
```

In the truncated sorted summary, we can see that there is some improvement with the new options:

```
Sorted summary from all runs:
Flags                                          Build     Test     Time
-------------------------------              -------   -------  ------
-O3 -OPT:unroll_times_max=8 -CG:load_          PASS      PASS     10.33
exe=0 -LNO:interchange=off
-CG:local_fwd_sched=on

-O3 -OPT:unroll_times_max=8 -CG:load_          PASS      PASS     10.45
exe=0 -LNO:interchange=off
-OPT:unroll_times_max=16

-O3 -OPT:unroll_times_max=8                    PASS      PASS     10.47

-O3 -OPT:unroll_times_max=8                    PASS      PASS     10.47
```

7.9.8.4 Example 3-Using a single script with the rate-file

With some applications or benchmarks, it is more convenient to combine building and testing into one script. In this case, you *must* use the `-S timing-file|rate-file` feature, so that you don't use the combined compile and run time as your sorting criterion to find the best solutions. Sometimes, the options that produce the fastest executable take more compile time.

One advantage of using a single script is that it is easier to parameterize, and requires less editing. For example, you can pass in another benchmark executable name from the command line rather than having to edit the name in the `psc_test` script.

We will use `-S rate-file` this time rather than `timing-file`. The use of `rate-file` means that we need to use `grep/sed` commands in the script below that differ from those in `psc_test2` above.

You can copy the file `compile-go-rate` from `/opt/pathscale/share/pathopt2/examples` into your working directory. It is show here:

```
#!/bin/sh
cd ..
make clean
code=$1
size=$2
shift 2
make $code CLASS=$size FFLAGS="$*"
cd pathopt2
../bin/$code.$size > logs/$code.$size.txt
grep "Mop" logs/$code.$size.txt > secs.log
sed -e 's/ Mop\/s total      =                //' \
  secs.log > $PSC_METRIC_FILE
grep SUCCESSFUL logs/$1.$2.txt
```

Make the file executable and run `pathopt2`:

```
$  chmod +x compile-go-rate
$  pathopt2 -S rate-file -t try5 \
     ./compile-go-rate ft A @
```

```
Sorted summary from all runs:
Flags                Build  Test   Rate
-------------------  -----  -----  ------
-Ofast               PASS   PASS   662.60
-O3 -ipa             PASS   PASS   662.37
-O3                  PASS   PASS   655.03
-O3 -OPT:Ofast       PASS   PASS   654.30
-O2                  PASS   PASS   603.43
```

Since `-Ofast` produced the best results in the sorted summary, we can now try the target `peak_Ofast`.

```
$  pathopt2 -S rate-file -t peak_Ofast \
     ./compile-go-rate ft A @
```

A truncated listing of the output shows the top five results for this run:

```
Sorted summary from all runs:
Flags                                          Build     Test     Rate
-------------------------------              -------  -------  ------
-Ofast -CG:prefetch=off -CG:load_exe=0         PASS     PASS    702.72
-OPT:unroll_size=256

-Ofast -CG:prefetch=off -CG:load_exe=0         PASS     PASS    702.17

-Ofast -msse3 -CG:load_exe=0                   PASS     PASS    696.36
-LNO:interchange=off -OPT:unroll_size=256

-Ofast -CG:prefetch=off -msse                  PASS     PASS    695.08
-CG:load_exe=0
-LNO:interchange=off

-Ofast -msse3 -CG:load_exe=0                   PASS     PASS    694.48
-LNO:interchange=off
```

In a situation like this, with a near tie at the top, one would normally use the simpler flag set for production:

```
-Ofast -CG:prefetch=off -CG:load_exe=0
```

which can be shortened to:

```
-Ofast -CG:prefetch=off:load_exe=0
```

7.10 How did the compiler optimize my code?

Often you may want to know what the compiler did to optimize your code. There are several ways to generate a listing showing (by line number) what the compiler did to optimize a subroutine. Choose the one that seems most useful to you.

7.10.1 Using the `-S` flag

The `-S` flag can be a useful way to see what the compiler did, especially if you understand some assembly, but it is useful even if you don't. Here is an example, using the `STREAM` benchmark. First we compile `STREAM` with the `-S` flag:

```
$ pathcc -O3 stream_d.c -S
```

This produces a `stream_d.s` assembly file. In this file you can see sections of human-readable comments interspersed with sections of assembly code, that look something like this:

```
#<loop> Loop body line 118, nesting depth: 1, iterations: 250000
#<loop> unrolled 4 times
#<sched>
#<sched> Loop schedule length: 13 cycles (ignoring nested loops)
#<sched>
#<sched>  4 flops ( 15% of peak)
#<sched>  8 mem refs ( 30% of peak)
#<sched>  3 integer ops ( 11% of peak)
#<sched> 15 instructions ( 28% of peak)
#<sched>
#<freq>
#<freq> BB:60 frequency = 250000.00000 (heuristic)
#<freq> BB:60 => BB:60 probability = 0.99994
#<freq> BB:60 => BB:59 probability = 0.00006
#<freq>
        .loc 1 120 0
# 119 for (j = 0; j < N; j++)
# 120 a[j] = 2.0E0 * a[j];
      movapd 0(%r8),%xmm3    # [0] id:82 a+0x0
      movapd 16(%r8),%xmm2   # [1] id:82 a+0x0
      addpd %xmm3,%xmm3      # [4]
      addpd %xmm2,%xmm2      # [5]
      movapd 32(%r8),%xmm1   # [2] id:82 a+0x0
      movapd 48(%r8),%xmm0   # [3] id:82 a+0x0
      addpd %xmm1,%xmm1      # [6]
      addpd %xmm0,%xmm0      # [7]
      movntpd %xmm3,0(%r8)   # [9] id:83 a+0x0
      movntpd %xmm2,16(%r8)  # [10] id:83 a+0x0
      addq $64,%r8           # [8]
      movntpd %xmm1,-32(%r8) # [11] id:83 a+0x0
      cmpq %rbp,%r8          # [11]
      movntpd %xmm0,-16(%r8) # [12] id:83 a+0x0
      jle .LBB60_main        # [12]
```

Note the "unrolled 4 times" comment above and the original source in comments, which tell you what the compiler did, even if you can't read x86 assembly code.

7.10.2 Using -CLIST or -FLIST

You can use `-CLIST:=on` (for C codes) or `-FLIST:=on` for Fortran codes to see what the compiler is doing. On the same `STREAM` source code, compile with the `-CLIST` flag:

```
$ pathcc -O3 -CLIST:=ON -c stream_d.c
```

The output will look something like this:

```
/opt/pathscale/lib/2.3.99/be translates /tmp/ccI.16xQZJ into
  stream.w2c.h and stream.w2c.c, based on source stream.c
```

When you look at `stream_d.w2c.c` with an editor, you might see some pretty strange looking C code. In this case, there doesn't seem to be much optimizing going on, but in codes where LNO (Loop Nest Optimization) is more important, you would see a lot of the optimizations.

7.10.3 Verbose flags

You can also turn on verbose flags in LNO to see vectorization activity. You would do this with the `-LNO:simd_verbose` flag in the compile line:

```
$ pathcc -O3 -LNO:simd_verbose -c stream_d.c
```

The output might look something like this:

```
(stream_d.c:103) LOOP WAS VECTORIZED.
(stream_d.c:119) LOOP WAS VECTORIZED.
(stream_d.c:142) LOOP WAS VECTORIZED.
(stream_d.c:147) LOOP WAS VECTORIZED.
(stream_d.c:152) LOOP WAS VECTORIZED.
(stream_d.c:157) LOOP WAS VECTORIZED.
(stream_d.c:164) Nonvectorizable ops/non-unit stride.
   Loop was not vectorized.
(stream_d.c:211) Nonvectorizable ops/non-unit stride.
   Loop was not vectorized.
```

This would tell you more about what the compiler is doing with loops. You can also try the `-LNO:vintr_verbose` flag on the compile line:

```
$ pathcc -O3 -LNO:vintr_verbose -c stream_d.c
```

In this case the output doesn't tell you much. No output because there are no intrinsic functions to get vectorized in `STREAM`.

Chapter 8

Using OpenMP and Autoparallelization

The PathScale EKOPath Compiler Suite includes OpenMP and autoparallelization for Fortran and C/C++.

This implementation of OpenMP supplies parallel directives that comply with the OpenMP Application Program Interface (API) specification 2.5. Runtime libraries and environment variables are also included. This chapter is *not* a tutorial on how to use OpenMP. To learn more about using OpenMP, please see a reference like *Parallel Programming in OpenMP*[1]. See Section 8.15 for more resources.

8.1 OpenMP

The OpenMP API defines compiler directives and library routines that make it relatively easy to create programs for shared memory computers (processors that share physical memory) from new or existing code. OpenMP provides a portable/scalable interface that has become the de facto standard for programming shared memory computers. Using OpenMP you can create threads, assign work to threads, and manage data within the program.

OpenMP enables incremental parallelization of your code on SMP (shared memory processor) systems, allowing you to add directives to chunks of existing code a little at a time.

The EKOPath OpenMP implementation in Fortran and C/C++ consists of parallelization directives and libraries. Using directives, you can distribute the work of the application over several processors.

OpenMP supports the three basic aspects of parallel programming: Specifying parallel execution, communicating between multiple threads, and expressing synchronization between threads.

[1] *Parallel Programming in OpenMP* by Rohit Chandra, et al; Morgan Kaufmann Publishers, 2000. ISBN 1-55-860671-8

The OpenMP runtime library automatically creates the optimal number of threads to be executed in parallel for the multiple processors on the platform where the program is being run. If you are running the program on a system with only one processor, you will not see any speedup. In fact, the program may run slower due to the overhead in the synchronization code generated by the compiler. For best performance, the number of threads should typically be equal to the number of processors you will be using.

The amount of speedup you can get under parallel execution depends a great deal on the algorithms used and the way the OpenMP directives are used. Programs that exhibit a high degree of coarse grain parallelism can achieve significant speedup as the number of processors are increased.

NOTE: OpenMP with certain C++ constructs is not supported. We recommend that C++ OpenMP programs be compiled with `-fno-exceptions`. Compiling for C++ applications that require both OpenMP and C++ exceptions is not currently supported. In addition, C++ OpenMP applications using C++ class data structures or class templates are not supported. An application that does not satisfy these restrictions can cause compile-time failure or runtime failure.

Appendix B describes the implementation dependent behavior for PathScale's OpenMP in C/C++ and Fortran. For more information on OpenMP and the OpenMP specification, please see the OpenMP website at `http://www.openmp.org`.

8.2 Autoparallelization

Under autoparallelization, the compiler tries to parallelize program code without depending on user directives. Autoparallization is invoked by specifying the `-apo` option on the compile and link lines:

```
$ pathf95 ... -apo .... -c foo.F95
$ pathf95 ... -apo .... -o foobar foo.o bar.o ...
```

Since the compiler is only able to parallelize a subset of the loops that the user knows are parallelizable, OpenMP directives are always helpful. OpenMP directives are not seen by the compiler unless `-mp` is specified. Thus, for programs that contain OpenMP directives, autoparallelization can be combined with OpenMP to additionally parallelize code that does not contain OpenMP directives. In this case it is good to specify the `-apo` and `-mp` options together.

```
$ pathf95 ... -apo -mp .... -c foo.F95
$ pathf95 ... -apo -mp .... -o foobar foo.o bar.o ...
```

Other than the OpenMP directives, the compiler currently does not implement any additional directives to help the compiler in its autoparallelization analysis.

Many codes benefit from autoparallelization and the extent of the benefit may vary with the characteristics of the program and data set being used. There are cases where autoparallelization causes small performance degradation of an

application. This happens because an autoparallelized program runs under multiple threads. The runtime decision to create multiple threads, followed by their synchronization, are overhead during execution.

When the compiler parallelizes a loop, it generates both a serial and a parallel version. At runtime, the generated code looks at the total amount of work performed by the loop and decides whether to execute the serial or the parallel version. This decision can only be made at runtime when the number of processors and the loop iteration counts are available. If the amount of work is not large enough to justify the additional synchronization overhead, it will execute the serial version instead. In such cases, the performance will be slower than if the program is not compiled with `-apo`, due to the need to make this decision at run time.

The synchronization overhead can be controlled using the `-LNO:parallel_overhead` option. The value of this option is the compiler's estimate of the overhead in processor cycles in invoking the parallel version of a loop. This value affects the runtime decision on whether to execute the serial or parallel versions. Because the optimal value varies across systems and programs, this option can be used for parallel performance tuning under `-apo`. For more information on this option, see the `eko` man page.

8.3 Getting started with OpenMP

To use OpenMP, you need to add directives where appropriate, and then compile and link your code using the `-mp` flag. This flag tells the compiler to honor the OpenMP directives in the program and process the source code guarded by the OpenMP conditional compilation sentinels (e.g. `!$` for Fortran and `#pragma` for C/C++). The actual program execution is also affected by the way the OpenMP Environment Variables (see Section 8.9) are set.

The compiler will generate different output that causes the program to be run in multiple threads during execution. The output code is linked with the PathScale OpenMP Runtime Library for execution under multiple threads. See the Fortran code in Section 8.12 and the C/C++ code in 8.13 for examples.

Because the OpenMP directives tell the compiler what constructs in the program can be parallelized, and how to parallelize them, it is possible to make mistakes in the inserted OpenMP code that will result in incorrect execution. As long as all the OpenMP-related code is guarded by conditional compilation sentinels (e.g. `!$` or `#pragma`), you can re-compile the same program without the `-mp` flag. In these cases, the resulting executable will run serially. If the error no longer occurs, you can conclude that the problems in the parallel execution are due to mistakes in the OpenMP part of the code, making the problem easier to track down and fix.

See Section 10.10 for more tips on troubleshooting OpenMP problems.

8.4 OpenMP compiler directives (Fortran)

The OpenMP directives for Fortran all start with comment characters followed by `$OMP` or `$omp`. They are only processed by the compiler if `-mp` is specified.

NOTE: Possible comment characters that can be used include `!`, `C`, `c`, and `*`. In the following examples we use `!` as the comment character. The Open MP standard dictates that for fixed-form Fortran, `!$OMP` directives must begin in the first column of the line.

Some of the OpenMP directives also support additional clauses. The following table lists the Fortran compiler directives provided by version 2.0 of the OpenMP Fortran Application Program Interface.

Fortran Compiler Directives		
Directive	**Clauses**	**Example**
Parallel region construct Defines a parallel region		
PARALLEL		!$OMP parallel [clause] ... structured-block !$OMP end parallel
	PRIVATE	
	SHARED	
	DEFAULT (FIRSTPRIVATE/ SHARED/ NONE)	
	REDUCTION	
	COPYIN	
	IF	
	NUM_THREADS	
Work sharing constructs Divide the execution of the enclosed block of code among the members of the team that encounter it		
DO	(NOWAIT)	!$OMP do [clause] ... do-loop !$OMP enddo [nowait]
	PRIVATE	
	FIRSTPRIVATE	
	LASTPRIVATE	
	REDUCTION	
	SCHEDULE (static, dynamic, guided, runtime)	
	ORDERED	
SECTIONS		!$OMP sections [clause]... structured-block !$OMP end sections [nowait]
	PRIVATE	
	FIRSTPRIVATE	
	LASTPRIVATE	
	REDUCTION	
SINGLE		!$OMP single [clause]... structured-block !$OMP end single [nowait]
	PRIVATE	
	FIRSTPRIVATE	
	COPYPRIVATE	
Combined parallel work sharing constructs Shortcut for denoting a parallel region that contains only one work-sharing construct		
PARALLEL DO		!$OMP parallel do structured-block !$OMP end parallel do

Fortran Compiler Directives		
Directive	**Clauses**	**Example**
PARALLEL SECTIONS		!$OMP parallel sections structured-block !$OMP end parallel sections
PARALLEL WORKSHARE		!$OMP parallel workshare structured-block !$OMP end parallel workshare
Synchronization constructs Provide various aspects of synchronization; for example, access to a block of code or execution order of statements within a block of code		
ATOMIC		!$OMP atomic expression-statement
BARRIER		!$OMP barrier
CRITICAL		!$OMP critical [(name)] structured-block !$OMP end critical [(name)]
FLUSH		!$OMP flush [(list)]
MASTER		!$OMP master structured-block !$OMP end master
ORDERED		!$OMP ordered structured-block !$OMP end ordered
Data environments Control the data environment during the execution of parallel constructs		
THREADPRIVATE		!$OMP threadprivate (/c1/, /c2/)
WORKSHARE		!$OMP workshare

8.5 OpenMP compiler directives (C/C++)

The OpenMP directives for C and C++ all start with `#pragma`. They are only processed by the compiler if `-mp` is specified.

Some of the OpenMP directives also support additional clauses. The following table lists the C and C++ compiler directives provided by version 2.0 of the OpenMP C/C++ Application Program Interface.

C/C++ Compiler Directives		
Directive	**Clauses**	**Example**
Parallel region construct Defines a parallel region		
PARALLEL		#pragma omp parallel [clause] ... structured-block
	PRIVATE	
	SHARED	
	FIRSTPRIVATE DEFAULT (SHARED/ NONE)	
	REDUCTION	
	COPYIN	
	IF	
	NUM_THREADS	
Work sharing constructs Divide the execution of the enclosed block of code among the members of the team that encounter it		

C/C++ Compiler Directives		
Directive	**Clauses**	**Example**
FOR	NOWAIT	`#pragma omp for [clause] ...` `for-loop`
	PRIVATE	
	FIRSTPRIVATE	
	LASTPRIVATE	
	REDUCTION	
	SCHEDULE (static, dynamic, guided, runtime)	
	ORDERED	
SECTIONS	NOWAIT	`#pragma omp sections [clause]...` `structured-block`
	PRIVATE	
	FIRSTPRIVATE	
	LASTPRIVATE	
	REDUCTION	
SINGLE	NOWAIT	`#pragma omp single [clause]...` `structured-block`
	PRIVATE	
	FIRSTPRIVATE	
	COPYPRIVATE	
Combined parallel work sharing constructs Shortcut for denoting a parallel region that contains only one work-sharing construct		
PARALLEL FOR		`#pragma omp parallel for` `structured-block`
PARALLEL SECTIONS		`#pragma omp parallel sections` `structured-block`
Synchronization constructs Provide various aspects of synchronization; for example, access to a block of code or execution order of statements within a block of code		
ATOMIC		`#pragma omp atomic` `expression-statement`
BARRIER		`#pragma omp barrier`
CRITICAL		`#pragma omp critical [(name)]` `structured-block`
FLUSH		`#pragma omp flush [(list)]`
MASTER		`#pragma omp master` `structured-block`
ORDERED		`#pragma omp ordered` `structured-block`
Data environments Control the data environment during the execution of parallel constructs		
THREADPRIVATE		`#pragma omp threadprivate`

8.6 OpenMP runtime library calls (Fortran)

OpenMP programs can explicitly call standard routines implemented in the OpenMP runtime library. If you want to ensure the program is still compilable without `-mp`, you need to guard such code with the OpenMP conditional compilation sentinels (e.g. `!$`). The following table lists the OpenMP runtime library routines provided by version 2.0 of the OpenMP Fortran Application Program Interface.

Fortran OpenMP Runtime Library Routines	
Routine	Description
`call omp_set_num_threads (integer)`	Set the number of threads to use in a team.
`integer omp_get_num_threads ()`	Return the number of threads in the currently executing parallel region.
`integer omp_get_max_threads ()`	Return the maximum value that `omp_get_num_threads` may return.
`integer omp_get_thread_num ()`	Return the thread number within the team.
`integer omp_get_num_procs ()`	Return the number of processors available to the program.
`call omp_set_dynamic (logical)`	Control the dynamic adjustment of the number of parallel threads.
`logical omp_get_dynamic ()`	Return `.TRUE.` if dynamic threads is enabled, otherwise return `.FALSE.`
`logical omp_in_parallel ()`	Return `.TRUE.` for calls within a parallel region, otherwise return `.FALSE.`
`call omp_set_nested (logical)`	Enable or disable nested parallelism.
`logical omp_get_nested ()`	Return `.TRUE.` if nested parallelism is enabled, otherwise return `.FALSE.`
Lock routines	
`omp_init_lock (int)`	Allocate and initialize lock, associating it with the lock variable passed in as a parameter.
`omp_init_nest_lock (int)`	Initialize a nestable lock and associate it with a specified lock variable.
`omp_set_lock (int)`	Acquire the lock, waiting until it becomes available if necessary.
`omp_set_nest_lock (int)`	Set a nestable lock. The thread executing the subroutine will wait until a lock becomes available and then set that lock, incrementing the nesting count.
`omp_unset_lock (int)`	Release the lock, resuming a waiting thread (if any).
`omp_unset_nest_lock (int)`	Release ownership of a nestable lock. The subroutine decrements the nesting count and releases the associated thread from ownership of the nestable lock.
`logical omp_test_lock (int)`	Try to acquire the lock, return `TRUE` if successful, `FALSE` if not.
`omp_test_nest_lock (int)`	Attempt to set a lock using the same method as `omp_set_nest_lock` but execution thread does not wait for confirmation that the lock is available. If lock is successfully set, function increments the nesting count, if lock is unavailable, function returns a value of zero.
`omp_get_wtime`	Returns double precision value equal to the number of seconds since the initial value of the operating system real-time clock.
`omp_get_wtick`	Returns double precision floating point value equal to the number of seconds between successive clock ticks.

8.7 OpenMP runtime library calls (C/C++)

OpenMP programs can explicitly call standard routines implemented in the OpenMP runtime library. If you want to ensure the program is still compilable without `-mp`, you need to guard such code with the OpenMP conditional compilation sentinels (e.g. `#pragma`). The following table lists the OpenMP runtime library routines provided by version 2.1 of the OpenMP C/C++ Application Program Interface.

C/C++ OpenMP Runtime Library Routines	
Routine	Description
`void omp_set_num_threads (int)`	Set the number of threads to use in a team.
`int omp_get_num_threads (void)`	Return the number of threads in the currently executing parallel region.
`int omp_get_max_threads (void)`	Return the maximum value that `omp_get_num_threads` may return.
`int omp_get_thread_num (void)`	Return the thread number within the team.
`int omp_get_num_procs (void)`	Return the number of processors available to the program.
`void omp_set_dynamic (int)`	Control the dynamic adjustment of the number of parallel threads.
`int omp_get_dynamic (void)`	Return `a non-zero value` if dynamic threads is enabled, otherwise return `0.`
`int omp_in_parallel (void)`	Return `a non-zero value` for calls within a parallel region, otherwise return `0.`
`void omp_set_nested (int)`	Enable or disable nested parallelism.
`int omp_get_nested (void)`	Return `a non-zero value` if nested parallelism is enabled, otherwise return `0.`
Lock routines	
`omp_init_lock (omp_lock_t *)`	Allocate and initialize lock, associating it with the lock variable passed in as a parameter.
`omp_init_nest_lock (omp_nest_lock_t *)`	Initialize a nestable lock and associate it with a specified lock variable.
`omp_set_lock (omp_lock_t *)`	Acquire the lock, waiting until it becomes available if necessary.
`omp_set_nest_lock (omp_nest_lock_t *)`	Set a nestable lock. The thread executing the subroutine will wait until a lock becomes available and then set that lock, incrementing the nesting count.
`omp_unset_lock (omp_lock_t *)`	Release the lock, resuming a waiting thread (if any).
`omp_unset_nest_lock (omp_nest_lock_t *)`	Release ownership of a nestable lock. The subroutine decrements the nesting count and releases the associated thread from ownership of the nestable lock.
`int omp_test_lock (omp_lock_t *)`	Try to acquire the lock, return `a non-zero value` if successful, `0` if not.
`omp_test_nest_lock (omp_nest_lock_t *)`	Attempt to set a lock using the same method as `omp_set_nest_lock` but execution thread does not wait for confirmation that the lock is available. If lock is successfully set, function increments the nesting count and returns the new nesting count, if lock is unavailable, function returns a value of zero.

C/C++ OpenMP Runtime Library Routines	
Routine	Description
`double omp_get_wtime (void)`	Returns double precision value equal to the number of seconds since the initial value of the operating system real-time clock.
`double omp_get_wtick (void)`	Returns double precision floating point value equal to the number of seconds between successive clock ticks.

8.8 Runtime libraries

There are both static and dynamic versions of each library, and the libraries are supplied in both 64-bit and 32-bit versions.

The libraries are:

`/opt/pathscale/lib/<`*version*`>/libopenmp.so`
- dynamic 64-bit

`/opt/pathscale/lib/<`*version*`>/libopenmp.a`
- static 64-bit

`/opt/pathscale/lib/<`*version*`>/32/libopenmp.so`
- dynamic 32-bit

`/opt/pathscale/lib/<`*version*`>/32/libopenmp.a`
- static 32-bit

The symbolic links to the dynamic versions of the libraries, for both 32-bit and 64-bit environments can be found here:

`/opt/pathscale/lib/<version>/libopenmp.so.1`
- symbolic link to dynamic version, 64-bit

`/opt/pathscale/lib/<`*version*`>/32/libopenmp.so.1`
- symbolic link to dynamic version, 32-bit

Be sure to use the `-mp` flag on both the compile and link lines.

NOTE: For running OpenMP executables compiled with the EKOPath compiler, on a system where no EKOPath compiler is currently installed, please see the *EKOPath Compiler Suite Install Guide,* Section 3.8: “Runtime installation” for instructions on installing the EKOPath libraries on the target system.

8.9 Environment variables

The OpenMP environment variables allow you to change the execution behavior of the program running under multiple threads. The table in this section lists the environment variables currently supported.

The environment variables can be set using the shell commands.

For example, in `bash`:

```
export OMP_NUM_THREADS=4
```

In csh:

```
setenv OMP_NUM_THREADS 4
```

After the previous shell commands, the following command will print 4:

```
echo $OMP_NUM_THREADS
4
```

Section 8.9.1 lists the available environment variables (both Standard and PathScale) for use with OpenMP.

8.9.1 Standard OpenMP environment variables

Standard OpenMP environment variables		
Variable	**Possible Values**	**Description**
OMP_DYNAMIC	FALSE	Enables or disables dynamic adjustment of the number of threads available for execution. Default is FALSE, since this mechanism is not supported.
OMP_NESTED	TRUE OR FALSE	Enables or disables nested parallelism. Default is FALSE.
OMP_SCHEDULE	*type* [, *chunk*]	This environment variable only applies to DO and PARALLEL_DO directives that have schedule type RUNTIME. Type can be STATIC, DYNAMIC, or GUIDED. Default is STATIC, with no chunk size specified.
OMP_NUM_THREADS	Integer value	Set the number of threads to use during execution. Default is number of CPUs in the machine.

8.9.2 PathScale OpenMP environment variables

PSC_OMP_AFFINITY (TRUE or FALSE)

When TRUE, the operating system's affinity mechanism (where available) is used to assign threads to CPUs, otherwise no affinity assignments are made. The default value is TRUE.

PSC_OMP_AFFINITY_GLOBAL (boolean TRUE or FALSE)

This environment variable controls where thread global ID or local ID values are used when assigning threads to CPUs. The default is TRUE so that global ID values are used for calculating thread assignments.

Global IDs uniquely identify each thread, and are integer values starting from 0 (for the original master thread) and incrementing upwards in the order in which

threads are allocated. The global ID is constant for a particular thread from its fork to its join. Using the global ID for the affinity mapping ensures that threads do not change CPU in their lifetime, and ensures that threads will be evenly distributed over CPUs.

The alternative is to use the thread local ID for this mapping. When nested parallelism is not employed, then each thread's global and local ID will be identical and the setting of this variable is irrelevant. However, when a nested team of threads is created, that team will be assigned new local thread IDs starting at `0` for the master of that team and incrementing upwards. Note that the local ID of a thread can change when that thread performs a nested fork and then a nested join, and that these events may cause the CPU binding of that thread to change. Also note that all team masters will have a local ID of `0`, and will therefore map to the same CPU. Usually these properties are undesirable, so the default is to use the thread global ID for scheduling assignments.

`PSC_OMP_AFFINITY_MAP` (a list of integer values separated by commas)

This environment variable allows the mapping from threads to CPUs to be fully specified by the user. It must be set to a list of CPU identifiers separated by commas. The list must contain at least one CPU identifier, and entries in the list beyond the maximum number of threads supported by the implementation (256) are ignored. Each CPU identifier is a decimal number between 0 and one less than the number of CPUs in the system (inclusive).

The implementation generates a mapping table that enumerates the mapping from each thread to CPUs. The CPU identifiers in the `PSC_OMP_AFFINITY_MAP` list are inserted in the mapping table starting at the index for thread `0` and increasing upwards. If the list is shorter than the maximum number of threads, then it is simply repeated over and over again until there is a mapping for each thread. This repeat feature allows short lists to be used to specify repetitive thread mappings for all threads.

Here are some examples for assigning eight threads on an eight CPU system:

1. Assign all threads to the same CPU: `PSC_OMP_AFFINITY_MAP=0`

CPU 0	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5	CPU 6	CPU 7
T0							
T1							
T2							
T3							
T4							
T5							
T6							
T7							

2. Assign threads to the lower half of the machine:
`PSC_OMP_AFFINITY_MAP=0,1,2,3`

CPU 0	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5	CPU 6	CPU 7
T0	T1	T2	T3				
T4	T5	T6	T7				

3. Assign threads to the upper half of the machine:
`PSC_OMP_AFFINITY_MAP=4,5,6,7`

CPU 0	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5	CPU 6	CPU 7
				T0	T1	T2	T3
				T4	T5	T6	T7

4. Assign threads to a dual-core machine in the same way as `PSC_OMP_CPU_STRIDE=2`:
`PSC_OMP_AFFINITY_MAP=0,2,4,6,1,3,5,7`

CPU 0	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5	CPU 6	CPU 7
T0	T4	T1	T5	T2	T6	T3	T7

NOTE: When `PSC_OMP_AFFINITY_MAP` is defined, the values of `PSC_OMP_CPU_STRIDE` and `PSC_OMP_CPU_OFFSET` are ignored. However, the value of `PSC_OMP_GLOBAL_AFFINITY` still determines whether the thread's global or local ID is used in the mapping process.

`PSC_OMP_CPU_STRIDE` `(Integer value)`

This specifies the striding factor used when mapping threads to CPUs. It takes an integer value in the range of 0 to the number of CPUs (inclusive). The default is a stride of 1 which causes the threads to be linearly mapped to consecutive CPUs. When there are more threads than CPUs the mapping wraps around giving a round-robin allocation of threads to CPUs. The behavior for a stride of 0 is the same as a stride of 1.

Strides greater than 1 are useful when there is a hierarchy of CPUs in the system, and the scheduling algorithm needs to take account of this to make best use of system resources. A particularly interesting case is when the system comprises a number of multi-core chips, such that each core shares some resources (e.g. a memory interface) with other cores on that chip. It may then be desirable to spread threads across the chips first to make best use of that resource, before scheduling multiple threads to the cores on each chip.

Let the number of CPUs in a multi-core chip be `m`, and the number of multi-core chips in the system be `n`. The total number of CPUs is then `n` multiplied by `m`. There are two typical orders in which the system may number the CPUs:

- For chip index `p` in `[0, n)` and core index `c` in `[0, m)`, the CPU number is `p*m + c`. This is *core-major* ordering as the core number varies fastest.
- For chip index `p` in `[0, n)` and core index `c` in `[0, m)`, the CPU number is `p + c*n`. This is *chip-major* ordering as the chip number varies fastest.

For *chip-major* ordering, a linear assignment of threads to CPU numbers will have the effect of spreading threads over chips first. For *core-major* ordering, the linear assignment will fill up the first chip with threads, before moving to the second chip, and so forth. This behavior can be changed by setting the stride factor to the value of `m`. It causes the OpenMP library to spread the threads across the CPUs with a stride equal to the number of cores in a chip.

For example, here are the generated thread assignments for a system comprising of four chips, each with two cores, where `PSC_OMP_CPU_STRIDE` is set to `2`:

<– CHIP 0 –>		<– CHIP 1 –>		<– CHIP 2 –>		<– CHIP 3 –>	
CPU 0	**CPU 1**	**CPU 2**	**CPU 3**	**CPU4**	**CPU 5**	**CPU 6**	**CPU 7**
T0	T4	T1	T5	T2	T6	T3	T7
T8	T12	T9	T13	T10	T14	T11	T15
T16	...						

`Tx` indicates thread number `x`. Here is another example for two chips with four cores and `PSC_OMP_CPU_STRIDE` set to `4`:

<———- CHIP 0 ———>				<———- CHIP 1 ———>			
CPU 0	**CPU 1**	**CPU 2**	**CPU 3**	**CPU 4**	**CPU 6**	**CPU 6**	**CPU 7**
T0	T2	T4	T6	T1	T3	T5	T7
T8	T10	T12	T14	T9	T11	T13	T15
T16	...						

This variable is most useful when the number of threads is fewer than the number of CPUs. In the common case where the number of threads is the same as the number of CPUs, then there is typically no need to set `PSC_OMP_CPU_STRIDE`.

Note that the same mappings can also be obtained by enumerating the CPU numbers using the `PSC_OMP_AFFINITY_MAP` variable.

`PSC_OMP_CPU_OFFSET` `(Integer value)`

This specifies an integer value that is used to offset the CPU assignments for the set of threads. It takes an integer value in the range of 0 to the number of CPUs (inclusive). When a thread is mapped to a CPU, this offset is added onto the CPU number calculated after `PSC_OMP_CPU_STRIDE` has been applied. If the resulting value is greater than the number of CPUs, then the remainder is used from the division of this value by the number of CPUs.

The effect of this is to apply an offset to the CPU assignments for a set of threads. This is particularly useful when multiple OpenMP jobs are to be run at the same time on the same system, and allows the jobs to be separated onto different CPUs. Without this mechanism both jobs would be assigned to CPUs starting at `CPU 0` causing a non-uniform distribution.

For example, consider a system with four chips each with two cores using core-major numbering. Let there be 2 OpenMP jobs each consisting of 4 threads. If these jobs are run with the default scheduling the assignments will be:

<– CHIP 0 –>		<– CHIP 1 –>		<– CHIP 2 –>		<– CHIP 3 –>	
CPU 0	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5	CPU 6	CPU 7
J0-T0	J0-T1	J0-T2	J0-T3				
J1-T0	J1-T1	J1-T2	J1-T3				

`Jx-Ty` indicates thread `y` of job `x`. If `PSC_OMP_CPU_OFFSET` is set to `4` for job `1`, the scheduling will be changed to:

<– CHIP 0 –>		<– CHIP 1 –>		<– CHIP 2 –>		<– CHIP 3 –>	
CPU 0	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5	CPU 6	CPU 7
J0-T0	J0-T1	J0-T2	J0-T3	J1-T0	J1-T1	J1-T2	J1-T3

If `PSC_OMP_CPU_STRIDE` is set to `2` for both jobs and `PSC_OMP_CPU_OFFSET` is set to `1` for job `1` only then the scheduling will be:

<– CHIP 0 –>		<– CHIP 1 –>		<– CHIP 2 –>		<– CHIP 3 –>	
CPU 0	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5	CPU 6	CPU 7
J0-T0	J1-T0	J0-T1	J1-T1	J0-T2	J1-T2	J0-T3	J1-T3

PSC_OMP_GUARD_SIZE (Integer value)

This environment variable specifies the size in bytes of a guard area that is placed below pthread stacks. This guard area is in addition to any guard pages created by your O/S. It is often useful to have a larger guard area to catch pthread stack overflows, particularly for Fortran OpenMP programs. By default, the guard area size is 0 for 32-bit programs (disabling the mechanism) and 32MB for 64-bit programs (since virtual memory is typically bountiful in 64-bit environments). The PSC_OMP_GUARD_SIZE environment variable can be used to over-ride the default value. Its format is a decimal number following by an optional 'k', 'm' or 'g' (in lower or uppercase) to denote kilobytes, megabytes, or gigabytes. If the size is 0 then the guard is not created. The guard area consumes no physical memory, but does consume virtual memory and will show up in the "VIRT" or "SIZE" figure of a "top" command.

PSC_OMP_GUIDED_CHUNK_DIVISOR (Integer value)

The value of PSC_OMP_GUIDED_CHUNK_DIVISOR is used to divide down the chunk size assigned by the guided scheduling algorithm. If the number of iterations left to be scheduled is remaining_size and the number of threads in the team is number_of_threads, the chunk size will be determined as:

```
chunk_size = (remaining_size) / (number_of_threads *
   PSC_OMP_GUIDED_CHUNK_DIVISOR)
```

A value of 1 gives the biggest possible chunks and the fewest number of calls into the loop scheduler. Larger values will result in smaller chunks giving more opportunities for the dynamic guided scheduler to assign work, balancing out variation between loop iterations, at the expense of more calls into the loop scheduler. With a value of PSC_OMP_GUIDED_CHUNK_DIVISOR equal to 1, the first thread will get 1/n'th of the iterations (for a team of n). If these iterations happen to be particularly expensive then this thread will be the critical path through the loop. The default value is 2.

PSC_OMP_GUIDED_CHUNK_MAX (Integer value)

This is the maximum chunk size that will be used by the loop scheduler for guided scheduling. The default value for this is 300. Note that a minimum chunk size can already be set by the user on a guided schedule directive. This environment variable allows the user to set a maximum too (though it applies to the whole program). The rationale for setting a maximum is to break up the iterations under guided scheduling for better dynamic load balancing between the threads.

The full equation for the chunk size for guided scheduling is:

```
chunk_size = MAX(
                MIN(
                    ROUNDUP(
                      (remaining_size) /
                      (number_of_threads
```

```
                    * PSC_OMP_GUIDED_CHUNK_DIVISOR)
                ),
                PSC_OMP_GUIDED_CHUNK_MAX
            ),
            minimum_chunk_size
        )
```

Where:

- `remaining_size` is the number of iterations of the loop.
- `number_of_threads` is the number of threads in the team.
- `PSC_OMP_GUIDED_CHUNK_DIVISOR` is the value of the `PSC_OMP_GUIDED_CHUNK_DIVISOR` environment variable (defaults to `2`).
- `PSC_OMP_GUIDED_CHUNK_MAX` is the value of the `PSC_OMP_GUIDED_CHUNK_MAX` environment variable (defaults to `300`).
- `minimum_chunk_size` is the size of the smallest piece (this is the value of chunk in the `SCHEDULE` directive)
- `ROUNDUP(x)` rounds `x` upwards to the nearest higher integer
- `MIN(a,b)` is the minimum of `a` and `b`
- `MAX(a,b)` is the maximum of `a` and `b`

The `minimum_chunk_size` is the value specified by the user in the guided scheduling directive (defaults to `1`).

NOTE: If the values of `PSC_OMP_GUIDED_CHUNK_MAX` and `minimum_chunk_size` are inconsistent (i.e. the minimum is larger than the maximum), the `minimum_chunk_size` takes precedence per the OpenMP specification.

`PSC_OMP_LOCK_SPIN` `(Integer value (0 or non-zero))`

This chooses the locking mechanism used by critical sections and OMP locks:

0 = user-level spin locks are disabled, uses `pthread` mutexes

non-zero = user-level spin locks are enabled. This is the default.

This determines whether locking in critical sections and OMP locks is implemented with user-level spin loops or using `pthread` mutexes. Synchronization using `pthread` mutexes is significantly more expensive but frees up execution resources for other threads.

`PSC_OMP_SILENT` `(Set or not set)`

If you set `PSC_OMP_SILENT` to anything, then warning and debug messages from the `libopenmp` library are inhibited. Fatal error messages are not affected by the setting of `PSC_OMP_SILENT`.

PSC_OMP_STACK_SIZE (Stack size specifications)

Stack size specification follows the syntax in Section 3.11. See Section 8.10.1 for more details.

PSC_OMP_STATIC_FAIR (Set or not set)

The default static scheduling policy when no chunk size is specified is as follows. The number of iterations of the loop is divided by the number of threads in the team and rounded **up** to give the chunk size. Loop iterations are grouped into chunks of this size and assigned to threads in order of increasing thread id (within the team). If the division was not exact then the last thread will have fewer iterations, and possibly none at all.

The policy for static scheduling when no chunk size is specified can be changed to the "static fair" policy by defining the environment variable PSC_OMP_STATIC_FAIR. The number of iterations is divided by the number of threads in the team and rounded **down** to give the chunk size. Each thread will be assigned at least this many iterations. If the division was not exact then the remaining iterations are scheduled across the threads in increasing thread order until no more iterations are left. The set of iterations assigned to a thread are always contiguous in terms of their loop iteration value. Note that the difference between the minimum and maximum number of iterations assigned to individual threads in the team is at most 1. Thus, the set of iterations is shared as fairly as possibly among the threads.

Consider the static scheduling of four iterations across 3 threads. With the default policy threads 0 and 1 will be assigned two iterations and thread 2 will be assigned no iterations. With the fair policy, thread 0 will be assigned two iterations and threads 1 and 2 will be assigned one iteration.

NOTE: The maximum number of iterations assigned to a thread (which determines the worst case path through the schedule) is the same for the default scheduling policy and the fair scheduling policy. In many cases the performance of these two scheduling policies will be very similar.

PSC_OMP_THREAD_SPIN (Integer value)

This takes a numeric value and sets the number of times that the spin loops will spin at user-level before falling back to O/S schedule/reschedule mechanisms. By default it is 100. If there are more active threads than processors and this is set very high, then the thread contention will typically cause a performance drop. Synchronization using the O/S schedule and reschedule mechanisms is significantly more expensive but frees up execution resources for other threads.

8.10 OpenMP stack size

8.10.1 Stack size for Fortran

The Fortran compiler allocates data on the stack by default. Some environments set a low limit on the size of a process' stack, which may cause Fortran programs that use a large amount of data to crash shortly after they start. In an OpenMP program there is a stack for the main thread of execution as in serial programs,

and also an additional separate stack for each additional thread created by `libopenmp`. These additional threads are created by the `POSIX` threads library and are called `pthreads` The PathScale EKOPath Fortran runtime environment automatically sizes the stack for the main thread and the `pthreads` to avoid stack size problems where possible. Additionally, diagnostics are given on memory segmentation faults to help diagnose stack size issues.

The stack size limit for the main thread of an OpenMP program is set using the same algorithm as for a serial Fortran program (see Section 3.11 for information about Fortran compiler stack size) except that the calculated stack limit is subsequently divided by the number of CPUs in the system. This ensures that the physical memory available for stack can be shared between as many threads as there are CPUs in the system. The limit tries to avoid excessive swapping in the case where all of these threads consume all of their available stack. Note that if there are more OpenMP threads than CPUs and they all consume all of their stack, then this will cause swapping. The stack size of the main thread can be controlled using the `PSC_STACK_LIMIT` environment variable, and diagnostics for its setting can be generated using the `PSC_STACK_VERBOSE` environment variable, in exactly the same way as for a serial Fortran program.

The stack sizing of OpenMP `pthreads` follows a complementary approach to that for the main thread. There are some differences because the sizing of `pthread` stacks has different system imposed limits and mechanisms. The `PSC_STACK_VERBOSE` flag can also be used to turn on diagnostics for the stack sizing of `pthreads`. However, the stack size is controlled by the `PSC_OMP_STACK_SIZE` environment variable (not `PSC_STACK_LIMIT`). The syntax and allowed values for `PSC_OMP_STACK_SIZE` are identical to the `PSC_STACK_LIMIT` so please see Section 3.11 for instructions.

The reason for having both `PSC_OMP_STACK_LIMIT` and `PSC_OMP_STACK_SIZE` is to allow the stacks of the main thread and the OpenMP `pthreads` to have different limits. Often, the system imposed limits are different in these two cases and sometimes the stack requirements of the OpenMP `pthreads` may be quite different from the main thread. For example, in some applications the main thread of an OpenMP program might allocate large arrays for the whole program on its stack, and in others the large arrays will be allocated by all of the threads.

8.10.2 Stack size for C/C++

The stack size of serial C and C++ programs is typically set by the `ulimit` command provided by the shell. Since C and C++ programs typically do not allocate large arrays on the stack it is usually convenient to use whatever default `ulimit` your system provides. More strict `ulimit` settings can be used to catch runaway stacks or unbounded recursion before the program exhausts all available memory.

For OpenMP C and C++ programs, there will be an additional stack for each `pthread` created by the `libopenmp` library. Section 8.11 describes how these `pthread` stacks are sized.

NOTE: The automatic stack sizing algorithm used by Fortran serial program and Fortran OpenMP programs is not employed for C and C++ programs.

8.11 Stack size algorithm

The stack limit for each OpenMP `pthread` is calculated as follows:

- If `PSC_OMP_STACK_SIZE` is set then this specifies the stack limit.
- If this is a Fortran program the stack limit is automatically set using the same approach as described in Section 3.11, except that the calculated value is divided by the number of CPUs in the system. This ensures that the physical memory available for stack can be shared between as many threads as there are CPUs in the system.
- Otherwise, this is a C/C++ program and the stack limit is set to a default value of 32MB. The distinction between Fortran and C/C++ programs is determined by whether the program entry point is `MAIN` (for Fortran) or `main` (for C/C++).

This stack size is then compared against system imposed limits (both lower and upper). If the check fails then a warning is generated, and the stack size is automatically adjusted to the appropriate limit. The following lower limit is imposed:

- The minimum size of a `pthread` stack specified by the system. This is typically 16KB.

The following upper limits are imposed:

- The maximum stack size that the system's `pthread` library will accept (i.e. the system-imposed upper bound on the `pthread` stack size). The library dynamically detects this value at start-up time. For systems using `linuxthreads`, this limit is typically in the range of 8MB to 32MB. For systems using `NPTL` threads, there is typically no arbitrary limit imposed by the system on the stack size.
- `libopenmp` imposes a limit of 1GB is imposed when using the 32-bit version of `libopenmp`, and a limit of 4GB when using the 64-bit version of `libopenmp`. These limits prevent excessive stack limits when using `libopenmp`.

When each `pthread` is created, the operating system will allocate virtual memory for its entire stack (as sized by the above algorithms). This essentially allocates virtual memory space for that stack so that it can grow up to its specified limit. The operating system will provide physical memory pages to back up this virtual memory as and when it is required. A consequence for this is that the "`top`" program will include the whole of these stacks in the `VIRT` or `SIZE`[2] memory usage figure, while only the allocated physical pages for these stacks will be shown in the `RES` or `RSS`[3] (resident) figure. If the OpenMP program runs with a large `pthread` stack size (which is the common case), then it is quite normal for `VIRT` or `SIZE` to be a large figure. It will be at least the number of `pthreads`

[2] `VIRT` or `SIZE` will be used depending on your Linux distribution.
[3] `RES` or `RSS` will be used depending on your Linux distribution.

created by libopenmp times their stack size. However, RES or RSS will typically be much less and this is the real physical memory requirement for the application.

NOTE: A large stack limit for the main thread does not show up in the VIRT or SIZE figure. This is because the operating system has special handling for the main thread of an application and does not need to pre-allocate virtual memory pages for its stack up to the stack limit.

The pthread stack limit is typically much lower when using linuxthreads than with NPTL threads. Linux kernels in the 2.4 series (and earlier) tend to be provided with linuxthreads, while NPTL is typically the default with 2.6 series kernels. However, some distributions have back-ported NPTL to their 2.4 series kernels.

NOTE: When a program is statically linked with pthreads this might also trigger use of linuxthreadson some distributions.

For best libopenmp performance and to avoid stack size limitations, it is highly recommended that 2.6 series Linux kernels, NPTL and dynamic linkage is used with OpenMP programs.

8.12 Example OpenMP code in Fortran

The following program is a parallel version of hello world written using OpenMP directives. When run, it spawns multiple threads. It uses the CRITICAL directive to ensure that the printing from the various threads will not overwrite one another.

Here is the program omphello.f:

```
      PROGRAM HELLO
      INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM
      TID=0
      NTHREADS=1
!     Fork a team of threads giving them their own copies of variables TID
!$OMP PARALLEL PRIVATE(TID)
!     Obtain and print thread id
!$    TID = OMP_GET_THREAD_NUM()
!$OMP CRITICAL
      PRINT *, 'Hello World from thread ', TID
!$OMP END CRITICAL
!$OMP MASTER
!$OMP CRITICAL
!     Only master thread does this
!$    NTHREADS= OMP_GET_NUM_THREADS()
      PRINT *, 'Number of threads = ', NTHREADS
!$OMP END CRITICAL
!$OMP END MASTER
!     All threads join master thread and disband
!$OMP END PARALLEL
      END
```

The `!$` before some of the lines are conditional compilation tokens. These lines are ignored when compiled without `-mp`.

We compile `omphello.f` for OpenMP with this command:

```
$ pathf95 -c -mp omphello.f
```

Now we link it, again using `-mp`:

```
$ pathf95 -mp omphello.o -o omphello.out
```

We set the environment variable for the number of threads with this command:

```
$ export OMP_NUM_THREADS=5
```

Now run the program:

```
$ ./omphello.out
 Hello World from thread  1
 Hello World from thread  2
 Hello World from thread  3
 Hello World from thread  0
 Number of threads =  5
 Hello World from thread  4
```

The output from the different threads can be in a different order each time the program is run. We can change the environment variable to run with two threads:

```
$ export OMP_NUM_THREADS=2
```

Now the output looks like this:

```
$ ./omphello.out
Hello World from thread 0
Number of threads = 2
Hello World from thread 1
```

The same program can be compiled and linked without `-mp` and the directives will be ignored. We compile the program (without `-mp`):

```
$ pathf95 -c omphello.f
```

Link the object file and create an output file:

```
$ pathf95 omphello.o -o omphello.out
```

Run the program and the output looks like this:

```
$ ./omphello.out
Hello World from thread 0
Number of threads = 1
```

For more examples using OpenMP, please see the sample code at `http://www.openmp.org/drupal/node/view/14`. There are also examples of OpenMP code in Appendix A of the OpenMP 2.0 Fortran specification. See Section 8.15 for more details.

8.13 Example OpenMP code in C/C++

The following program is a parallel version of `hello world` written using OpenMP directives. When run, it spawns multiple threads. It uses the `CRITICAL` directive to ensure that the printing from the various threads will not overwrite one another.

Here is the program `omphello.c`:

```
#include <omp.h>
main()
{
  int tid = 0;
  int nthreads = 1;
/* Fork a team of threads giving them their own copies of variable tid */
  #pragma omp parallel private (tid)
  {
#ifdef _OPENMP
    /* Obtain and print thread id */
    tid = omp_get_thread_num ();
#endif
    #pragma omp critical
      printf ("Hello World from thread %d\n", tid);
    #pragma omp master
    #pragma omp critical
    {
#ifdef _OPENMP
      /* Only master thread does this */
      nthreads = omp_get_num_threads ();
#endif
      printf ("Number of threads = %d\n", nthreads);
    }
    /* All threads join master thread and disband */
  }
}
```

The `#pragma` and `#ifdef` before some of the lines are conditional compilation tokens. These lines are ignored when compiled without `-mp`.

We compile `omphello.c` for OpenMP with this command:

```
$ pathcc -c -mp omphello.c
```

Now we link it, again using `-mp`:

```
$ pathcc -mp omphello.o -o omphello.out
```

We set the environment variable for the number of threads with this command:

```
$ export OMP_NUM_THREADS=5
```

Now run the program:

```
$ ./omphello.out
 Hello World from thread  1
 Hello World from thread  2
 Hello World from thread  3
 Hello World from thread  0
 Number of threads =  5
 Hello World from thread  4
```

The output from the different threads can be in a different order each time the program is run. We can change the environment variable to run with two threads:

```
$ export OMP_NUM_THREADS=2
```

Now the output looks like this:

```
$ ./omphello.out
Hello World from thread 0
Number of threads = 2
Hello World from thread 1
```

The same program can be compiled and linked without `-mp` and the directives will be ignored. We compile the program (without `-mp`):

```
$ pathcc -c omphello.c
```

Link the object file and create an output file:

```
$ pathcc omphello.o -o omphello.out
```

Run the program and the output looks like this:

```
$ ./omphello.out
Hello World from thread 0
Number of threads = 1
```

For more examples using OpenMP, please see the sample code at `http://www.openmp.org/drupal/node/view/14`. There are also examples of OpenMP code in Appendix A of the OpenMP 2.0 C/C++ specification. See Section 8.15 for more details.

8.14 Tuning for OpenMP application performance

A good first step in tuning OpenMP code is to build a serial version of the application and tune the serial performance (See Chapter 7 for ideas and suggestions). Often good flags for serial performance are also good for OpenMP performance. Typically OpenMP parallelizes the outer iterations of the compute intensive loops in a coarse fashion, leaving chunks of the outer loops and the inner loops that generally behave very similarly to the serial code.

Use `pathopt` (see Section 7.9 for details on `pathopt`) to help find good serial tuning options for the application. You may be able to find interesting options for tuning by looking at tuned configuration files for similar codes.

With this approach you can find good options for the serial parts of the code before having to consider OpenMP-specific issues (such as scheduling, scaling, and affinity). If the test case takes a long time to run or needs a lot of memory, then you may be forced to tune the flags with OpenMP enabled.

8.14.1 Reduced datasets

You may find it useful to reduce the size of the data sets to give a quicker runtime, allowing the efficacy of particular tuning options to be quickly ascertained. One thing to note is that OpenMP performance tends to get better with larger data sets because the fork/join overheads diminish as the loops get larger. Thus, you should also run trials with the full data set, especially when looking at scaling issues. You can also make use of more memory and more cache on an `n`-way multi-processor than a uni-processor, and this sometimes leads to a very nice superlinear speed-up.

8.14.2 Enable OpenMP

After you have tuned the serial version of the application, turn on OpenMP parallelization with the `-mp` flag. Try running the code on varying numbers of CPUs to see how the application scales.

One very important option for OpenMP tuning is `-OPT:early_mp,` which by default is off but can be turned on using `-OPT:early_mp=on`. The setting of this primarily determines the ordering of (SIMD) vectorization and OpenMP parallelization optimization phase of the compiler. With late MP, loops will first be vectorized and then the vectorized loops will be parallelized. With early MP loops will first be parallelized and then the parallel loops will be vectorized. Occasionally one of these orderings works better than the other, so you have to try both.

8.14.3 Optimizations for OpenMP

The most important optimizations for OpenMP applications tend to be loop nest optimization (LNO), code generation (CG) and aggressive optimizations (e.g. by

reducing numerical accuracy). IPA (inter-procedural analysis) may help with OpenMP programs too–try it and see!

NOTE: Currently it is not possible to use feedback directed optimization (FDO) with OpenMP programs.

8.14.3.1 Libraries

Some applications spend a large amount of time in numerical libraries. At small numbers of nodes, a highly optimized and tuned serial algorithm crafted for the target processor may out perform a parallel implementation based on a non-optimized algorithm. At higher numbers of nodes the parallel version may scale and give better performance. However, best performance will typically require an OpenMP parallelization of the best serial algorithm (exploiting target features such as SSE for example). Check to see if there are OpenMP-enabled versions of these numerical libraries available.

8.14.3.2 Memory system performance

OpenMP applications are often very sensitive to memory system performance. An excellent approach is to tune the memory system with an OpenMP version of the `STREAM` benchmark. In particular, the BIOS settings for memory bank interleaving should be `auto`, and for node interleaving should be `off`.

Interleaving memory by node causes memory addresses to be striped across the various nodes at a low granularity, creating the illusion of a uniform memory system. However, OpenMP programs tend to have very good memory locality and the correct approach is to use NUMA optimizations in the operating system to give good placement of data relative to threads.

This optimization relies on first touch: the thread that first touches the data is assumed to be the most frequent user of the data and thus the data is allocated onto physical addresses in the DRAM associated with the CPU that is currently running that thread. This is applied by a NUMA-aware operating system at the page level. If your kernel version is not NUMA aware, then a kernel upgrade may be required for good performance.

Similarly thread-to-CPU affinity is also important for good OpenMP performance. The OpenMP library by default uses affinity system calls to strongly associate threads with CPUs. The idea is to keep the threads co-located with their associated data. Without affinity assignments, the threads may be migrated by the O/S scheduler to other nodes and lose their good placement relative to their data. However, sometimes the use of affinity binding can cause a load imbalance and prevent the scheduler from make sensible decisions about thread placement. In this case the thread affinity assignments can be disabled by setting the `PSC_OMP_AFFINITY` environment variable to `FALSE`. If your kernel does not support scheduling affinity, you may need to upgrade to a newer kernel to see the performance benefit of this mechanism.

8.14.3.3 Load balancing

It is possible to gain some coarse insight into the load balancing of the OpenMP application using the "`top`" program. Depending on the version of "`top`", you should be able to view the breakdown of user, system, and idle time per CPU. Often this view can be obtained by pushing "`1`". You may also want to increase the update rate (e.g with "`s`" followed by `0.5`). It is sometimes possible to see the program moving from serial to parallel phases and also see whether the work is being well distributed. If there is excessive time spent in the system or swapping, then this should also be investigated. It goes without saying that it is best to run OpenMP applications on nodes with no other running applications.

If the OpenMP application uses runtime scheduling, then try varying the runtime schedule using the `OMP_SCHEDULE` environment variable. A good choice of schedule and chunk size is sometimes important for performance.

NOTE: The `gprof` profiling (`-pg`) does not work in conjunction with `pthreads` or the OpenMP library. An alternative approach is to use OProfile, which uses hardware counters and sampling techniques to build up a profile of the system.

It is possible to capture application code, dynamic libraries, kernel, modules, and drivers in a profile created by OProfile giving insight into system-wide performance characteristics. OProfile can also attribute the samples on a thread or CPU basis allowing load balancing and scheduling issues to be observed. OProfile can access many different performance counters giving more detail insight into the CPU behavior; however, these advanced features of OProfile are not easy to use.

If the application uses nested OpenMP parallelism, then try turning on the nested parallelism support by setting the `OMP_NESTED` environment variable to `TRUE`.

8.14.3.4 Tuning the application code

If you are able to tune the code of the application, it is worth checking whether any of the OpenMP directives specify a chunk size. It may be possible to make more appropriate choices of the chunk size, perhaps influenced by the number of CPUs available, the L2 size, or the data size. You may also want to try different scheduling strategies. If the amount of work in an OpenMP loop varies significantly from iteration to iteration, then a `DYNAMIC` or `GUIDED` scheduling algorithm is preferable.

The default loop scheduling algorithm is static scheduling and this is used by the majority OpenMP applications. If this leads to an unbalanced distribution of work across the threads, try setting the `PSC_OMP_STATIC_FAIR` environment variable, which will cause the library to use a fairer distribution.

If the application uses guided scheduling, the `PSC_OMP_GUIDED_CHUNK_DIVISOR` and `PSC_OMP_GUIDED_CHUNK_MAX` environment variables can be used to tune the loop scheduling. The default values for these are widely applicable but some applications with guided scheduling can be fairly sensitive to their setting. See Section 8.9.2 for the interpretation of these.

By default the OpenMP library employs spin locks for synchronization and these loops can be tuned for performance using the `PSC_OMP_THREAD_SPIN` and

PSC_OMP_LOCK_SPIN environment variables. It may be desirable to turn off the spinning (and use blocking pthread calls instead) for OpenMP applications that use multiple threads per CPU. This is fairly uncommon, and in the usual case the use of spin locks is a significant optimization over the use of blocking pthread calls. (See Section 8.9.2 for details on these environment variables.)

8.14.3.5 Using feedback data

If an OpenMP program is instrumented via the -fb-create option to generate feedback data in feedback-directed compilation, the execution of the instrumented executable should only be run under a single thread. This can be effected via the OMP_NUM_THREADS environment variable. The reason is because the instrumentation library (libinstr.so) used during execution does not support simultaneous updates of the feedback data by multiple threads. Running the instrumented executable under multiple threads can result in segmentation faults.

8.15 Other resources for OpenMP

For more information on OpenMP, you might also find these resources useful:

- At the OpenMP home page, http://www.openmp.org/
  - For the Fortran version 2.0 OpenMP Specification, click on *Specifications* in the left column of the OpenMP home page
  - For Tutorials, Benchmarks, Publications, and Books, click on *Resources* in the left column of the OpenMP home page.
- *Parallel Programming in OpenMP* by Rohit Chandra, et al; Morgan Kaufmann Publishers, 2000. ISBN 1-55-860671-8

Chapter 9

Examples

9.1 Compiler flag tuning and profiling with `pathprof`

We'll use the `168.wupwise` program from the CPU2000 floating point suite for this example. This is a Physics/Quantum Chromodynamics (QCD) code. For those who care, "wupwise" is an acronym for "Wuppertal Wilson Fermion Solver," a program in the area of lattice gauge theory (quantum chromodynamics). The code is in about 2100 lines of Fortran 77 in 23 files. We'll be running and tuning `wupwise` performance on the reference (largest) dataset. Each run takes about two to four minutes on a 2 GHz Opteron system to complete.

Even though this is a Fortran 77 code, the PathScale EKOPath Fortran compiler (`pathf95`) can handle it.

Outline:

Try `pathf95 -O2` and `pathf95 -O3` first.

Run times (user time) were:

```
          seconds
-O2        150.3
-O3        174.3
```

We're a little surprised since `-O3` is supposed to be faster than `-O2` in general. But the man page did say that the `-O3` "may include optimizations that are generally beneficial but may hurt performance."

So, let's look at a profile of the `-O2` binary. We do need to recompile using flags `-O2 -pg`.

Then we need to run the generated, instrumented binary again with the same reference dataset: `$ time -p ./wupwise > wupwise.out` (Here we used the `-p` (`POSIX`) flag to get a different time output format). This run generates the file `gmon.out` of profiling information.

Then we need to run `pathprof` to generate the human-readable profile.

```
$ pathprof ./wupwise
Flat profile:
Each sample counts as 0.01 seconds.
 %   cumulative  self              self     total
time   seconds   seconds  calls  s/call s/call name
51.15     83.54    83.54 155648000     0.00     0.00  zgemm_
17.65    112.37    28.83 603648604     0.00     0.00  zaxpy_
 8.72    126.61    14.24 214528306     0.00     0.00  zcopy_
 8.03    139.72    13.11 933888000     0.00     0.00  lsame_
 4.59    147.21     7.49                             s_cmp
 1.51    149.67     2.46   512301     0.00     0.00  zdotc_
 1.49    152.11     2.44 603648604     0.00     0.00  dcabs1_
 1.37    154.34     2.23 155648000     0.00     0.00  gammul_
 1.08    156.10     1.76 155648000     0.00     0.00  su3mul_
 1.07    157.85     1.75      152     0.01     0.50  muldeo_
...
 0.00    163.32     0.00        1     0.00   155.83  MAIN__
 0.00    163.32     0.00        1     0.00     0.00  init_
 0.00    163.32     0.00        1     0.00     0.06  phinit_
 %     the percentage of the total running time of the
time   program used by this function.
cumulative   a running sum of the number of seconds accounted
seconds      for by this function and those listed above it.
...
```

NOTE: The `pathprof` program included in the PathScale EKOPath Compiler Suite is a symbolic link your system's `gprof` executible. The `pathprof` and `pathcov` programs link to the `gprof` and `gcov` executibles in the version of GCC on which the EKOPath Compiler Suite is based. Please note that the `pathprof` tool will generate a segmentation fault when used with OpenMP applications that are run with more than one thread. There is no current workaround for `pathprof` (or `gprof`).

Now, we note that the total time that `pathprof` measures is 163.3 secs. vs. the 150.3 that we measured for the original `-O2` binary. But considering that the `-O2 -pg` instrumented binary took 247 seconds to run, this is a pretty good estimate.

It is nice that the top hot-spot, `zgemm` consumes about 50% of the total time. We also note that some very small routines `zaxpy`, `zcopy`, and `lsame` are called a very large number of times. These look like ideal candidates for inlining.

In the second part of the `pathprof` output (after the explanation of the column headings for the flat profile), is a call-graph profile. In the example of such a profile below, one can follow the chain of calls from `main` to `matmul_`, `muldoe_`, `su3mul_`, and `zgemm_`, where most of the time is consumed.

```
================================================
Additional call-graph profile info:
                    Call graph (explanation follows)
   granularity: each sample hit covers 4 byte(s) for 0.01%
of 163.32 seconds
index % time    self  children    called     name
                0.00  155.83       1/1           main [2]
[1]     95.4    0.00  155.83       1         MAIN__ [1]
                0.00  151.19     152/152         matmul_ [3]
```

```
                0.05    4.47       1/1           uinith_ [13]
                0.00    0.06       1/1           phinit_ [22
                0.02    0.04       1/2           rndphi_ [21]
                0.00    0.00     301/512301      zdotc_ [14]
                0.00    0.00      77/1024077     dznrm2_ [17]
                0.00    0.00     452/603648604     zaxpy_ [9]
                0.00    0.00     154/214528306     zcopy_ [10]
                0.00    0.00      75/39936075     zscal_ [16]
                0.00    0.00       1/1           init_ [23]
-----------------------------------------------
  ...
 -----------------------------------------------
                0.00  151.19     152/152         MAIN__ [1]
[3]     92.6    0.00  151.19     152         matmul_ [3]
                1.75   73.84     152/152         muldoe_ [7]
                1.75   73.84     152/152         muldeo_ [6]
                0.00    0.00     152/214528306     zcopy_ [10]
                0.00    0.00     152/603648604     zaxpy_ [9]
-----------------------------------------------
                0.88   48.33 77824000/155648000    muldeo_ [6]
                0.88   48.33 77824000/155648000    muldoe_ [7]
[4]     60.3    1.76   96.65 155648000            su3mul_ [4]
               83.54   13.11 155648000/155648000   zgemm_ [5]
-----------------------------------------------
               83.54   13.11 155648000/155648000   su3mul_ [4]
[5]     59.2   83.54   13.11 155648000             zgemm_ [5]
               13.11    0.00 933888000/933888000   lsame_ [11]
-----------------------------------------------
  ...
================================================================
```

The `-ipa` option can analyze the code to make smart decisions on when and which routines to inline so we try that. `-O2 -ipa` results in a 133.8 second run time–a nice improvement over our previous best of 150 seconds with only `-O2`.

Since we heard somewhere that improvements with compiler flags are not always predictable, we also try `-O3 -ipa`. To our great surprise, we achieve a run time of 110.5 seconds, a 58% speed-up over our previous `-O3` time, and a nice improvement over `-O2 -ipa`.

Section 7.7 mentions the flags `-O3 -ipa -LNO:fusion=2` and `-OPT:div_split=on`. Testing combinations of these two flags as additions to the `-O3 -ipa` we have already tested results in:

`-O3 -ipa -LNO:fusion=2` results in 109.74 seconds run time
`-O3 -ipa -OPT:div_split=on` results in 112.24 seconds
`-O3 -ipa -OPT:div_split=on -LNO:fusion=2` results in 111.28 seconds

So, `-O3 -ipa` is essentially a tie for the best set of flags with `-O3 -ipa -LNO:fusion=2`.

Chapter 10

Debugging and troubleshooting

The *PathScale EKOPath Compiler Suite Support Guide* contains information about getting support from PathScale and tells you how to submit a bug. (We consider performance issues to be a bug.) The `pathbug` tool, described in the Support Guide, can help you gather information for submitting your bug.

10.1 Subscription Manager problems

For recommendations in addressing problems or issues with subscriptions, refer to ”Troubleshooting” in the *PathScale Subscription Management User Guide*.

10.2 Debugging

The earlier chapters on the PathScale EKOPath Fortran and C /C++ compilers contain language-specific debugging information. See Section 3.10 and Section 4.3. More general information on debugging can be found in this section.

The flag `-g` tells the PathScale EKOPath compilers to produce data in the form used by modern debuggers, such as `pathdb` or GDB. This format is known as DWARF 2.0 and is incorporated directly into the object files. Code that has been compiled using `-g` will be capable of being debugged using `pathdb`, GDB, or other debuggers. See the *PathScale Debugger User Guide* for more information on using `pathdb`.

It is advisable to use the `-O0` level of optimization in conjunction with the `-g` flag, since code rearrangement and other optimizations can sometimes make debugging difficult. If `-g` is specified without an optimization level, then `-O0` is the default.

10.3 Dealing with uninitialized variables

Uninitialized variables may cause your program to crash or to produce incorrect results. New options have been added to help identify and deal with uninitialized variables in your code. These options are `-trapuv`, `-Wuninitialized`, and `-zerouv`.

The `-trapuv` option works by initializing local variables to NaN (floating point not-a-number) and setting the CPU to detect floating point calculations involving NaNs. *Floating point calculations* are operations such as `+`, `-`, `*`, `/`, `sin`, `sqrt`, `compare`, etc. If a NaN is detected the application will abort. Assignments are not considered floating point calculations, and so "`x=y`" doesn't trap even if `y` is NaN.

The `-trapuv` option affects local scalar and array variables and memory returned by `alloca()`. It does not affect the behavior of globals, memory allocated with `malloc()`, or Fortran common data. The option initializes integer variables to the bit pattern for floating point NaN (integers don't have NaNs). The CPU doesn't trap on these integer operands, although the NaN bit pattern will make the wrong result more obvious. This option is not supported under 32-bit ABI without SSE2.

The `-Wuninitialized` option warns about uninitialized automatic variables at compile time. `-Wno-uninitialized` tells the compiler not to warn about uninitialized automatic variables.

The new `-zerouv` option sets uninitialized variables in your code to zero at program runtime. Doing this will have a slight performance impact. This option affects local scalar and array variables and memory returned by `alloca()`. It does not affect the behavior of globals, memory allocated with `malloc()`, or Fortran common data.

10.4 Large object support

Statically allocated data (`.bss`) objects such as Fortran COMMON blocks and C variables with file scope are currently limited to 2GB in size. If the total size exceeds that, the compilation (without the `-mcmodel=medium` option) will likely fail with the message:

```
relocation truncated to fit: R_X86_64_PC32
```

For Fortran programs with only one `COMMON` block or with no `COMMON` blocks after the one that exceeds the 2GB limit, the program may compile and run correctly.

At higher optimization levels (`-O3`, `-Ofast`), `-OPT:reorg_common` is set to `ON` by default. This might split a `COMMON` block such that a block begins beyond the 2GB boundary. If a program builds correctly at `-O2` or below but fails at `-O3` or `-Ofast`, try adding `-OPT:reorg_common=OFF` to the flags. Alternatively, using the `-mcmodel=medium` option will allow this optimization.

10.5 More inputs than registers

The compiler will complain if an asm has more inputs than there are available CPU registers. For m32 (32-bit), the maximum number of asm inputs is seven (7). For m64 (64-bit), the maximum number is fifteen (15).

10.6 Linking with `libg2c`

When using Fortran with a Red Hat or Fedora Core system, you cannot link `libg2c` automatically. In order to link successfully against `libg2c` on a Red Hat or Fedora Core system, you should first install the appropriate `libf2c` library, then add a symlink in `/usr/lib64` or `/usr/lib` from `libg2c.so.0` to `libg2c.so`. This problem is due to a packaging issue with Red Hat's version of this library.

You will only need to take this step if you are linking against either the AMD Core Math Library (ACML) or Fortran object code that was compiled using the `g77` compiler.

10.7 Linking large object files

The PathScale EKOPath Compiler Suite does not support the linking or assembly of large object files on the x86 platform.

Earlier versions of the compiler (before 2.1) contained a bug that would truncate static data structures whose size exceeded four gigabytes. This sometimes caused a compilation error or generation of binaries that would crash or corrupt data at runtime. This bug has been fixed in the 2.1 release.

10.8 Using `-ipa` and `-Ofast`

When compiling with `-ipa`, the `.o` files that are created are not a regular `.o` files. IPA uses the `.o` files in its analysis of your program, and then does a second compilation using that information.

NOTE: When you are using `-ipa`, *all* the `.o` files have to have been compiled *with* `-ipa` for your compilation to be successful. Each archive (for example `libfoo.a`) must contain either `.o` files compiled with `-ipa` or `.o` files compiled without `-ipa`, but not both.

The requirement of `-ipa` may mean modifying Makefiles. If your Makefiles build libraries, and you wish this code to be built with `-ipa`, you will need to split these libraries into separate `*.o` files before linking.

By default, `-ipa` is turned on when you use `-Ofast`, so the caveats above apply to using `-Ofast` as well.

10.9 Tuning

Our compilers often optimize loops by eliminating the loop variable, and instead using a quantity related to the loop variable, called an "induction variable". If the induction variable overflows, the loop test will be incorrectly evaluated. This is a very rare circumstance. To see if this is causing your code to fail under optimization, try:

```
-OPT:wrap_around_unsafe_opt=OFF
```

10.10 Troubleshooting OpenMP

You must use the `-mp` flag when you compile code that contains OpenMP directives. If you do not use the `-mp` flag, the compiler will ignore the OpenMP directives and compile your code as if the directives were not there.

10.10.1 Compiling and linking with `-mp`

If a program compiled with `-mp` is linked and linked without the `-mp` flag, the linker will not link with the OpenMP library and the linker will display undefined references similar to these:

```
: undefined reference to ‘__ompc_can_fork’
   ../libutil.a(diffu.o)(.text+0xa93): In function
   ‘diffu_’:
: undefined reference to ‘__ompc_get_thread_num’
   ../libutil.a(diffu.o)(.text+0x2400): In function
   ‘diffu_’:
: undefined reference to ‘__ompc_fork’
   ../libutil.a(diffu.o)(.text+0x2499): In function
   ‘__ompdo_diffu_1’:
```

Appendix A

Environment variables

This appendix lists environment variables utilized by the compiler, along with a short description. These variables are organized by language, with a separate section for language independent variables.

A.1 Environment variables for use with C

PSC_CFLAGS - Flags to pass to the the C compiler, `pathcc`. This variable is used with the `gcc` compatibility wrapper scripts.

A.2 Environment variables for use with C++

PSC_CXXFLAGS - Flags to pass to the C++ compiler, `pathCC`. This variable is used with the `gcc` compatibility wrapper scripts.

A.3 Environment variables for use with Fortran

F90_BOUNDS_CHECK_ABORT - Set to `YES`, causes the program to abort on the first bounds check violation.

F90_DUMP_MAP - Dump memory mapping at the location of a segmentation fault.

FTN_SUPPRESS_REPEATS - Output multiple values instead of using the repeat factor, used at runtime

NLSPATH - Flags for runtime and compile-time messages. If the `main` function in your program is coded in C, then even though other parts of the program are coded in Fortran, the Fortran runtime library will not be able to find the file which provides runtime error messages. To remedy this, set the `NLSPATH` environment variable to the location of the error messages, using `%N` for the base name of the file. For example, if the compiler version is 2.1, set it to `/opt/pathscale/lib/2.1/%N.cat`.

PSC_FDEBUG_ALLOC - Flag to debug Fortran memory allocations. This variable is used to initialize memory locations during execution.

PSC_FFLAGS - Flags to pass to the Fortran compiler, `pathf95`. This variable is used with the `gcc` compatibility wrapper scripts.

PSC_STACK_LIMIT - Controls the stack size limit the Fortran runtime attempts to use. This string takes the format of a floating-point number, optionally followed by one of the characters "`k`" (for units of 1024 bytes), "`m`" (for units of 1048576 bytes), "`g`" (for units of 1073741824 bytes), or "`%`" (to specify a percentage of physical memory). If the specifier is following by the string "`/cpu`", the limit is divided by the number of CPUs the system has. For example, a limit of "1.5g" specifies that the Fortran runtime will use no more than 1.5 gigabytes (GB) of stack. On a system with 2GB of physical memory, a limit of "90%/cpu" will use no more than 0.9GB of stack (2/2*0.90).

PSC_STACK_VERBOSE - If this environment variable is set, the Fortran runtime will print detailed information about how it is computing the stack size limit to use.

A.4 Language independent environment variables

FILENV - The location of the `assign` file. See the `assign` man page for more details.

PSC_COMPILER_DEFAULTS_PATH - Specifies a `PATH` or a colon-separated list of `PATH`s, designating where the compiler is to look for the `compiler.defaults` file. If the environment variable is set, the `PATH` `/opt/pathscale/etc` will not be used. If the file cannot be found, then no defaults file will be used, even if one is present in `/opt/pathscale/etc`.

PSC_GENFLAGS - Generic flags passed to all compilers. This variable is used with the `gcc` compatibility wrapper scripts.

PSC_PROBLEM_REPORT_DIR - Name a directory in which to save problem reports and preprocessed source files, if the compiler encounters an internal error. If not specified, the directory used is `$HOME/.ekopath-bugs`.

A.5 Environment variables for OpenMP

These environment variables are described in detail in Section **??** and **??**. They are listed here for your reference.

A.5.1 Standard OpenMP runtime environment variables

These environment variables can be used with OpenMP in either Fortran or C and C++.

OMP_DYNAMIC - Enables or disables dynamic adjustment of the number of threads available for execution. Default is `FALSE`, since this mechanism is not supported.

OMP_NESTED - Enables or disables nested parallelism. Default is FALSE.

OMP_SCHEDULE - This environment variable only applies to DO and PARALLEL_DO directives that have schedule type RUNTIME. Type can be STATIC, DYNAMIC, or GUIDED. Default is STATIC, with no chunk size specified.

OMP_NUM_THREADS - Set the number of threads to use during execution. Default is number of CPUs in the machine.

A.5.2 PathScale OpenMP environment variables

These environment variables can be used with OpenMP in Fortran and C and C++, except as indicated.

PSC_OMP_AFFINITY - When TRUE, the operating system's affinity mechanism (where available) is used to assign threads to CPUs, otherwise no affinity assignments are made. The default value is TRUE.

PSC_OMP_AFFINITY_GLOBAL - This environment variable controls where thread global ID or local ID values are used when assigning threads to CPUs. The default is TRUE so that global ID values are used for calculating thread assignments.

PSC_OMP_AFFINITY_MAP - This environment variable allows the mapping from threads to CPUs to be fully specified by the user. It must be set to a list of CPU identifiers separated by commas. The list must contain at least one CPU identifier, and entries in the list beyond the maximum number of threads supported by the implementation (256) are ignored. Each CPU identifier is a decimal number between 0 and one less than the number of CPUs in the system (inclusive).

The implementation generates a mapping table that enumerates the mapping from each thread to CPUs. The CPU identifiers in the PSC_OMP_AFFINITY_MAP list are inserted in the mapping table starting at the index for thread 0 and increasing upwards. If the list is shorter than the maximum number of threads, then it is simply repeated over and over again until there is a mapping for each thread. This repeat feature allows short lists to be used to specify repetitive thread mappings for all threads.

PSC_OMP_CPU_STRIDE - This specifies the striding factor used when mapping threads to CPUs. It takes an integer value in the range of 0 to the number of CPUs (inclusive). The default is a stride of 1 which causes the threads to be linearly mapped to consecutive CPUs. When there are more threads than CPUs the mapping wraps around giving a round-robin allocation of threads to CPUs. The behavior for a stride of 0 is the same as a stride of 1.

PSC_OMP_CPU_OFFSET - This specifies an integer value that is used to offset the CPU assignments for the set of threads. It takes an integer value in the range of 0 to the number of CPUs (inclusive). When a thread is mapped to a CPU, this offset is added onto the CPU number calculated after PSC_OMP_CPU_STRIDE has been applied. If the resulting value is greater than the number of CPUs, then the remainder is used from the division of this value by the number of CPUs.

PSC_OMP_GUARD_SIZE - This environment variable specifies the size in bytes of a guard area that is placed below pthread stacks. This guard area is in addition to any guard pages created by your O/S.

PSC_OMP_GUIDED_CHUNK_DIVISOR -The value of PSC_OMP_GUIDED_CHUNK_DIVISOR is used to divide down the chunk size assigned by the guided scheduling algorithm. See Section 8.9.2 for details.

PSC_OMP_GUIDED_CHUNK_MAX - This is the maximum chunk size that will be used by the loop scheduler for guided scheduling. See Section 8.9.2 for details.

PSC_OMP_LOCK_SPIN - This chooses the locking mechanism used by critical sections and OMP locks. See Section 8.9.2 for details.

PSC_OMP_SILENT - If you set PSC_OMP_SILENT to anything, then warning and debug messages from the libopenmp library are inhibited.

PSC_OMP_STACK_SIZE - (Fortran) Stack size specification follows the syntax in Section 3.11. See Section **??** for more details.

PSC_OMP_STATIC_FAIR - This determines the default static scheduling policy when no chunk size is specified, as discussed in Section 8.9.2.

PSC_OMP_THREAD_SPIN - This takes a numeric value and sets the number of times that the spin loops will spin at user-level before falling back to O/S schedule/reschedule mechanisms.

Appendix B

Implementation dependent behavior for OpenMP Fortran

The OpenMP Fortran specification 2.0, Appendix E, requires that the implementation defined behavior of PathScale's OpenMP implementation be defined and documented (see `http://www.openmp.org/`[1]). This appendix summarizes the behaviors that are described as implementation dependent in this API. The sections in italic, including the cross references, come from the Fortran 2.0 specification, and each is followed by the relevant details for the PathScale implementation in its EKOPath Compiler Suite 2.4 release of OpenMP for Fortran.

`SCHEDULE(GUIDED,chunk)`: `chunk` specifies the size of the smallest piece, except possibly the last. The size of the initial piece is implementation dependent (Table 1, page 17).

The size of the initial piece is given by the following equation:

```
chunk_size = MAX(
               MIN(
                 ROUNDUP(
                   (remaining_size) /
                   (number_of_threads * PSC_OMP_GUIDED_CHUNK_DIVISOR)
                 ),
                 PSC_OMP_GUIDED_CHUNK_MAX
               ),
               minimum_chunk_size
             )
```

Where:

- `remaining_size` is the number of iterations of the loop.

[1]For the Fortran version 2.0 OpenMP Specification, click on *Specifications* in the left column of the OpenMP home page.

- `number_of_threads` is the number of threads in the team.
- `PSC_OMP_GUIDED_CHUNK_DIVISOR` is the value of the `PSC_OMP_GUIDED_CHUNK_DIVISOR` environment variable (defaults to `2`).
- `PSC_OMP_GUIDED_CHUNK_MAX` is the value of the `PSC_OMP_GUIDED_CHUNK_MAX` environment variable (defaults to `300`).
- `minimum_chunk_size` is the size of the smallest piece (this is the value of chunk in the `SCHEDULE` directive)
- `ROUNDUP(x)` rounds `x` upwards to the nearest higher integer
- `MIN(a,b)` is the minimum of `a` and `b`
- `MAX(a,b)` is the maximum of `a` and `b`

When `SCHEDULE(RUNTIME)` is specified, the decision regarding scheduling is deferred until runtime. The schedule type and chunk size can be chosen at runtime by setting the `OMP_SCHEDULE` environment variable. If this environment variable is not set, the resulting schedule is implementation-dependent (Table 1, page 17).

The default runtime schedule is static scheduling. The default chunk size is set to the number of iterations of the loop divided by the number of threads in the team rounded up to the nearest integer. The loop iterations are partitioned into chunks of the default chunk size. If the number of iterations of the loop is not an exact integer multiple of the number of threads in the team, the last chunk will be smaller than the default chunk size and in some cases it may contain zero loop iterations. The chunks are assigned to threads starting from the thread with local index 0. The thread with the highest local index will receive the last chunk, and this may be smaller than the others or even zero. The loop iterations which are executed by a thread are contiguous in terms of their loop iteration number.

NOTE: The `PSC_OMP_STATIC_FAIR` environment variable can be used to change the default static scheduling algorithm to an alternate scheme where the iterations are more equally balanced over the threads in cases where the division in not exact.

In the absence of the `SCHEDULE` clause, the default schedule is implementation-dependent (Section 2.3.1, page 15).

In the absence of the `SCHEDULE` clause, the default schedule is static scheduling. The default chunk size is set to the number of iterations of the loop divided by the number of threads in the team rounded up to the nearest integer. The loop iterations are partitioned into chunks of the default chunk size. If the number of iterations of the loop is not an exact integer multiple of the number of threads in the team, the last chunk will be smaller than the default chunk size and in some cases it may contain zero loop iterations. The chunks are assigned to threads starting from the thread with local index 0. The thread with the highest local index will receive the last chunk, and this may be smaller than the others or even zero. The loop iterations which are executed by a thread are contiguous in terms of their loop iteration number.

NOTE: The `PSC_OMP_STATIC_FAIR` environment variable can be used to change the default static scheduling algorithm to an alternate scheme where the iterations are more equally balanced over the threads in cases where the division in not exact.

OMP_GET_NUM_THREADS: If the number of threads has not been explicitly set by the user, the default is implementation-dependent (Section 3.1.2, page 48).

If the number of threads has not been explicitly set by the user, it defaults to the number of CPUs in the machine.

OMP_SET_DYNAMIC: The default for dynamic thread adjustment is implementation-dependent (Section 3.1.7, page 51).

The default for OMP_DYNAMIC is false. Dynamic thread adjustment is not supported by this implementation–the number of threads that are assigned to a new team is not adjusted dynamically by this implementation.

If dynamic thread adjustment is requested by the user or program, by setting OMP_DYNAMIC to TRUE or calling OMP_SET_DYNAMIC with a TRUE parameter, the implementation produces a diagnostic message and ignores the request. The value returned by OMP_GET_DYNAMIC is always FALSE to indicate that this mechanism is not supported.

OMP_SET_NESTED: When nested parallelism is enabled, the number of threads used to execute nested parallel regions is implementation-dependent (Section 3.1.9, page 52).

The implementation supports dynamically-nested parallelism. The number of threads assigned to a new team is determined by the following algorithm:

- If this fork is dynamically nested inside another fork and nesting is disabled, then the new team will consist of 1 thread (the thread that requests the fork).
- Otherwise, the number of threads is specified by the NUM_THREADS clause on the parallel directive if NUM_THREADS has been specified.
- Otherwise, the number of threads is specified by the most recent call to OMP_SET_NUM_THREADS if it has been called.
- Otherwise, the number of threads is specified by the OMP_NUM_THREADS environment variable if it has been defined.
- Otherwise, the number of threads defaults to the number of CPUs in the machine.

If the number of threads is greater than 1, the request requires allocation of new threads and this may fail if insufficient machine resources are available. The maximum number of threads that can be allocated simultaneously is limited to 256 by the implementation.

Currently, nested parallelism is not supported where nested parallel directives are statically scoped within the same subroutine as the outer parallel directive. In this case only the outer parallel directive will be parallelized, and any inner nested directives will be serialized (executed by a team of 1 thread). To achieve nested parallelism, the nested parallel directives must be moved to a separate subroutine.

OMP_SCHEDULE environment variable: The default value for this environment variable is implementation-dependent (Section 4.1, page 59).

The default for the OMP_SCHEDULE environment variable is static scheduling with no chunk size specified. The chunk size will default to the number of iterations of the loop divided by the number of threads in the team rounded up to the nearest integer. The loop iterations are partitioned into chunks of the default chunk size. If the number of iterations of the loop is not an exact integer multiple of the number of threads in the team, the last chunk will be smaller than the default chunk size and in some cases it may contain zero loop iterations. The chunks are assigned to threads starting from the thread with local index 0. The thread with the highest local index will receive the last chunk, and this may be smaller than the others or even zero. The loop iterations which are executed by a thread are contiguous in terms of their loop iteration number.

NOTE: The PSC_OMP_STATIC_FAIR environment variable can be used to change the default static scheduling algorithm to an alternate scheme where the iterations are more equally balanced over the threads in cases where the division in not exact.

OMP_NUM_THREADS environment variable: The default value is implementation-dependent (Section 4.2, page 60).

The default value of the OMP_NUM_THREADS environment variable is the number of CPUs in the machine.

OMP_DYNAMIC environment variable: The default value is implementation-dependent (Section 4.3, page 60).

The default value of the OMP_DYNAMIC environment variable is false.

An implementation can replace all ATOMIC directives by enclosing the statement in a critical section (Section 2.5.4, page 27).

Many ATOMIC directives are implemented with in-line atomic code for the atomic statement, while others are implemented using a critical section, due to the absence of hardware support.

If the dynamic threads mechanism is enabled on entering a parallel region, the allocation status of an allocatable array that is not affected by a COPYIN clause that appears on the region is implementation-dependent (Section 2.6.1, page 32).

The allocation status of the thread's copy of an allocatable array will be retained on entering a parallel region.

Due to resource constraints, it is not possible for an implementation to document the maximum number of threads that can be created successfully during a program's execution. This number is dependent upon the load on the system, the amount of memory allocated by the program, and the amount of implementation dependent stack space allocated to each thread. If the dynamic threads mechanism is disabled, the behavior of the program is implementation-dependent when more threads are requested than can be successfully created. If the dynamic threads mechanism is enabled, requests for more threads than an implementation can support are satisfied by a smaller number of threads (Section 2.3.1, page 15).

Since the implementation does not support dynamic thread adjustment, the dynamic threads mechanism is always disabled. If more threads are requested than are available, the request will be satisfied using only the available threads.

The maximum number of threads that can be allocated simultaneously is limited to 256 by the implementation.

Additionally, if a system call to allocate threads, memory or other system resources does not succeed, then the runtime library will exit with a fatal error message.

If an `OMP` runtime library routine interface is defined to be generic by an implementation, use of arguments of kind other than those specified by the `OMP_*_KIND` constants is implementation-dependent (Section D.3, page 111).

No generic `OMP` runtime library routine interface is provided.

Appendix C

Supported Fortran intrinsics

The 2.4 release of the PathScale EKOPath Compiler Suite supports all of the GNU `g77` intrinsics. You must use `-intrinsic=PGI` or `-intrinsic=G77` to get new G77 intrinsics which were added in the 2.3 release.

All of the argument types for each intrinsic may not be supported in this release.

C.1 How to use the intrinsics table

As an example let's look at the intrinsic `ACOS`. This is what it looks like in the table:

Intrinsic Name	Result	Arguments	Families	Remarks
`ACOS`	R*4	X: R*4, R*8	ANSI, G77, PGI, TRADITIONAL	E, P

For the intrinsic `ACOS`, the result is `R*4`, which means "`REAL*4`" or "`REAL(KIND=4)`", and its arguments (`X`) can be either `R*4` (`REAL*4`) or `R*8` (`REAL*8`). `ACOS` belongs to the `ANSI`, `G77`, `PGI`, and `TRADITIONAL` families of intrinsics (see Section C.2 for an explanation of intrinsic families), which means the compiler will recognize it if any of those families is enabled. Under remarks, `E`, `P` are listed. `E` tells us that this is an elemental intrinsic and `P` tells us that the intrinsic may be passed as an actual argument.

Here is a simple scalar call to intrinsic `ACOS`:

```
print *, acos(1.0)
```

Because the intrinsic is elemental, you can also apply it to an array:

```
print *, acos((/ 1.0, 0.707, 0.5 /))
```

NOTE: One of the lesser-known features of Fortran 90 is that you can use argument names when calling intrinsics, instead of passing all of the arguments in strictly defined order. There are only a couple of cases where it is actually useful to know the official name so that you can omit optional arguments that don't interest you (for example `call date_and_time(time=timevar)`) but you're always allowed to specify the name if you like.

C.2 Intrinsic options

If your program contains a function or subroutine whose name conflicts with that of one of the intrinsic procedures, you have three choices. Within each program unit that calls that function or subroutine, you can declare the procedure in an "external" statement; or you can declare it with Fortran 90 interface block; or you can use command-line options to tell the compiler not to provide that intrinsic.

The option `-ansi` (if present) removes all non-standard intrinsics. The options `-intrinsic=`*name* and `-no-intrinsic=`*name* are applied to add or remove specific intrinsics from the set of remaining ones.

For example, the compile command might look like this:

```
$ pathf95 myprogram.f -ansi -intrinsic=second
```

To make it convenient to compile programs developed under other compilers, `pathf95` provides the ability to enable and disable a group or "family" of intrinsics with a single option. Family names are `ANSI`, `EVERY`, `G77`, `PGI`, `OMP`, and `TRADITIONAL`. These family names must appear in uppercase to distinguish them from the names of individual intrinsics. By default, the compiler enables either `ANSI` or `TRADITIONAL`, depending on whether you use the `-ansi` option. It automatically enables `OMP` as well if you use the `-mp` option.

As an example, suppose you are compiling a program that was originally developed under the GNU `G77` compiler, and encounter problems because it contains subroutine names which conflict with some of the intrinsics in the `TRADITIONAL` family. Suppose that you have also decided that you want to use the individual intrinsic `adjustl`, which is not provided by `G77`. These options would give you the set of intrinsics you need:

```
-no-intrinsic=TRADITIONAL -intrinsic=G77 -intrinsic=adjustl
```

C.3 Table of supported intrinsics

The following table lists the Fortran intrinsics supported by the PathScale EKOPath Compiler Suite, along with the result, arguments, families, and characteristics for each. See the **Legend** for more information.

Legend:

Key to Types	Key to Characteristics
I: Integer	**E**: Elemental intrinsic
R: Real	**P**: May pass intrinsic itself as an actual argument
Z: Complex	**X**: Extension to the Fortran standard
C: Character	**O**: Optional argument
L: Logical	
Depends on arg: Result type varies depending on the argument type	
Subroutine: Intrinsic is a subroutine, not a function	

Fortran Intrinsics Supported in 2.4

Intrinsic Name	Result	Arguments	Families	Remarks
ABORT	Subroutine		G77, PGI	
ABS	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		A: I*1, I*2, I*4, I*8, R*4, R*8, Z*8, Z*16		
ACCESS	I*4		G77, PGI	
		NAME: C		
		MODE: C		
ACHAR	C		ANSI, G77, PGI, TRADITIONAL	E
		I: I*1, I*2, I*4, I*8		
ACOS	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		X: R*4, R*8		
ACOSD	R*4		PGI, TRADITIONAL	E
		X: R*4, R*8		
ADD_AND_FETCH			TRADITIONAL	E
		I: I*4		
		J: I*4		
ADD_AND_FETCH			TRADITIONAL	E
		I: I*8		
		J: I*8		
ADJUSTL			ANSI, PGI, TRADITIONAL	E
		STRING: C		
ADJUSTR			ANSI, PGI, TRADITIONAL	E
		STRING: C		
AIMAG	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		Z: Z*8, Z*16		
AINT	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		A: R*4, R*8		
		KIND: I*1, I*2, I*4, I*8		O
ALARM	I*4		G77, PGI	
		SECONDS: I*4, I*8		
		HANDLER: Procedure		
		STATUS: I*4		O
ALARM	Subroutine		G77	
		SECONDS: I*4, I*8		
		HANDLER: Procedure		
		STATUS: I*4		O
ALL			ANSI, PGI, TRADITIONAL	See Std
ALLOCATED			ANSI, PGI, TRADITIONAL	See Std
ALOG	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		X: R*4, R*8		
ALOG10	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		X: R*4, R*8		

Intrinsic Name	Result	Arguments	Families	Remarks
AMAX0			ANSI, G77, PGI, TRADITIONAL	See Std
AMAX1			ANSI, G77, PGI, TRADITIONAL	See Std
AMIN0			ANSI, G77, PGI, TRADITIONAL	See Std
AMIN1			ANSI, G77, PGI, TRADITIONAL	See Std
AMOD	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		A: R*4, R*8		
		P: R*4, R*8		
AND			ANSI, G77, PGI, TRADITIONAL	E
		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8		
		J: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8		
AND_AND_FETCH			TRADITIONAL	E
		I: I*4		
		J: I*4		
AND_AND_FETCH			TRADITIONAL	E
		I: I*8		
		J: I*8		
ANINT	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		A: R*4, R*8		
		KIND: I*1, I*2, I*4, I*8		O
ANY			ANSI, PGI, TRADITIONAL	See Std
ASIN	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		X: R*4, R*8		
ASIND	R*4		PGI, TRADITIONAL	E
		X: R*4, R*8		
ASSOCIATED			ANSI, PGI, TRADITIONAL	See Std
ATAN	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		X: R*4, R*8		
ATAN2	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		Y: R*4, R*8		
		X: R*4, R*8		
ATAN2D	R*4		PGI, TRADITIONAL	E, P
		Y: R*4, R*8		
		X: R*4, R*8		
ATAND	R*4		PGI, TRADITIONAL	E, P
		X: R*4, R*8		
BESJ0	R*4		G77, PGI	
		X: R*4		

Intrinsic Name	Result	Arguments	Families	Remarks
BESJ0	R*8	X: R*8	G77, PGI	
BESJ1	R*4	X: R*4	G77, PGI	
BESJ1	R*8	X: R*8	G77, PGI	
BESJN	R*4	N: I*4 X: R*4	G77, PGI	
BESJN	R*8	N: I*4 X: R*8	G77, PGI	
BESY0	R*4	X: R*4	G77, PGI	
BESY0	R*8	X: R*8	G77, PGI	
BESY1	R*4	X: R*4	G77, PGI	
BESY1	R*8	X: R*8	G77, PGI	
BESYN	R*4	N: I*4 X: R*4	G77, PGI	
BESYN	R*8	N: I*4 X: R*8	G77, PGI	
BITEST		I: I*2 POS: I*1, I*2, I*4, I*8	PGI, TRADITIONAL	E
BIT_SIZE		I: I*1, I*2, I*4, I*8	ANSI, G77, PGI, TRADITIONAL	E
BJTEST		I: I*4 POS: I*1, I*2, I*4, I*8	PGI, TRADITIONAL	E
BKTEST		I: I*8 POS: I*1, I*2, I*4, I*8	TRADITIONAL	E
BTEST		I: I*1, I*2, I*4, I*8 POS: I*1, I*2, I*4, I*8	ANSI, G77, PGI, TRADITIONAL	E
CABS	R*4	A: Z*8, Z*16	ANSI, G77, PGI, TRADITIONAL	E, P
CCOS	Z*8	X: Z*8, Z*16	ANSI, G77, PGI, TRADITIONAL	E, P
CDABS	R*8	A: Z*16	G77, PGI, TRADITIONAL	E, P

Intrinsic Name	Result	Arguments	Families	Remarks
CDCOS	Z*16		G77, PGI, TRADITIONAL	E, P
		X: Z*16		
CDEXP	Z*16		G77, PGI, TRADITIONAL	E, P
		X: Z*16		
CDLOG	Z*16		G77, PGI, TRADITIONAL	E, P
		X: Z*16		
CDSIN	Z*16		G77, PGI, TRADITIONAL	E, P
		X: Z*16		
CDSQRT	Z*16		G77, PGI, TRADITIONAL	E, P
		X: Z*16		
CEILING			ANSI, PGI, TRADITIONAL	E
		A: R*4, R*8		
		KIND: I*1, I*2, I*4, I*8		O
CEXP	Z*8		ANSI, G77, PGI, TRADITIONAL	E, P
		X: Z*8, Z*16		
CHAR	C		ANSI, G77, PGI, TRADITIONAL	E
		I: I*1, I*2, I*4, I*8		
		KIND: I*1, I*2, I*4, I*8		O
CHDIR	I*4		G77, PGI	
		DIR: C		
		STATUS: I*4		O
CHDIR	Subroutine		G77	
		DIR: C		
		STATUS: I*4		O
CHMOD	I*4		G77, PGI	
		NAME: C		
		MODE: C		
		STATUS: I*4		O
CHMOD	Subroutine		G77	
		NAME: C		
		MODE: C		
		STATUS: I*4		O
CLEAR_IEEE_EXCEPTION	Subroutine		TRADITIONAL	E
		EXCEPTION: I*8		
CLOC	I*8		TRADITIONAL	
		C: C		
CLOCK	C		TRADITIONAL	
CLOG	Z*8		ANSI, G77, PGI, TRADITIONAL	E, P
		X: Z*8, Z*16		
CMPLX	Z*8		ANSI, G77, PGI, TRADITIONAL	E
		X: I*1, I*2, I*4, I*8, R*4, R*8, Z*8, Z*16		
		Y: I*1, I*2, I*4, I*8, R*4, R*8, Z*8, Z*16		O

Intrinsic Name	Result	Arguments	Families	Remarks
		KIND: I*1, I*2, I*4, I*8		O
COMMAND_ARGUMENT_COUNT	I*4		ANSI, TRADITIONAL	
COMPARE_AND_SWAP	L*4		TRADITIONAL	E
		I: I*4 J: I*4 K: I*4		
COMPARE_AND_SWAP	L*8		TRADITIONAL	E
		I: I*8 J: I*8 K: I*8		
COMPL			PGI, TRADITIONAL	E
		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8		
CONJG	Z*8		ANSI, G77, PGI, TRADITIONAL	E, P
		Z: Z*8, Z*16		
COS	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		X: R*4, R*8, Z*8, Z*16		
COSD	R*4		PGI, TRADITIONAL	E, P
		X: R*4, R*8		
COSH	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		X: R*4, R*8		
COT	R*4		TRADITIONAL	E, P
		X: R*4, R*8		
COTAN	R*4		TRADITIONAL	E, P
		X: R*4, R*8		
COUNT			ANSI, PGI, TRADITIONAL	See Std
CPU_TIME	Subroutine		ANSI, G77, PGI, TRADITIONAL	
		TIME: R*4		
CPU_TIME	Subroutine		ANSI, G77, PGI, TRADITIONAL	
		TIME: R*8		
CSHIFT			ANSI, PGI, TRADITIONAL	See Std
CSIN	Z*8		ANSI, G77, PGI, TRADITIONAL	E, P
		X: Z*8, Z*16		
CSMG			TRADITIONAL	E
		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 J: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 K: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8		
CSQRT	Z*8		ANSI, G77, PGI, TRADITIONAL	E, P
		X: Z*8, Z*16		

Intrinsic Name	Result	Arguments	Families	Remarks
CTIME	C	STIME: I*4	G77, PGI	
CTIME	C	STIME: I*8	G77, PGI	
CTIME	Subroutine	STIME: I*4 RESULT: C	G77	O
CTIME	Subroutine	STIME: I*8 RESULT: C	G77	O
CVMGM		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 J: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 K: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8	TRADITIONAL	E
CVMGN		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 J: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 K: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8	TRADITIONAL	E
CVMGP		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 J: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 K: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8	TRADITIONAL	E
CVMGT		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 J: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 K: L*1, L*2, L*4, L*8	TRADITIONAL	E
CVMGZ		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 J: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 K: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8	TRADITIONAL	E
C_LOC	I*8	X: Any type, Array rank=any	TRADITIONAL	
DABS	R*8	A: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DACOS	R*8	X: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DACOSD	R*8		PGI, TRADITIONAL	E

Intrinsic Name	Result	Arguments	Families	Remarks
		X: R*8		
DASIN	R*8		ANSI, G77, PGI, TRADITIONAL	E, P
		X: R*8		
DASIND	R*8		PGI, TRADITIONAL	E
		X: R*8		
DATAN	R*8		ANSI, G77, PGI, TRADITIONAL	E, P
		X: R*8		
DATAN2	R*8		ANSI, G77, PGI, TRADITIONAL	E, P
		Y: R*8		
		X: R*8		
DATAN2D	R*8		PGI, TRADITIONAL	E
		Y: R*8		
		X: R*8		
DATAND	R*8		PGI, TRADITIONAL	E
		X: R*8		
DATE	C		G77, PGI, TRADITIONAL	
DATE	Subroutine		G77, PGI	
		DATE: C		
DATE_AND_TIME	Subroutine		ANSI, G77, PGI, TRADITIONAL	
		DATE: C		O
		TIME: C		O
		ZONE: C		O
		VALUES: I*1, I*2, I*4, I*8, Array rank=1		O
DBESJ0	R*8		G77, PGI	
		X: R*8		
DBESJ1	R*8		G77, PGI	
		X: R*8		
DBESJN	R*8		G77, PGI	
		N: I*4		
		X: R*8		
DBESY0	R*8		G77, PGI	
		X: R*8		
DBESY1	R*8		G77, PGI	
		X: R*8		
DBESYN	R*8		G77, PGI	
		N: I*4		
		X: R*8		
DBLE	R*8		ANSI, G77, PGI, TRADITIONAL	E
		A: I*1, I*2, I*4, I*8, R*4, R*8, Z*8, Z*16		
DCMPLX	Z*16		G77, PGI, TRADITIONAL	E
		X: I*1, I*2, I*4, I*8, R*4, R*8, Z*8, Z*16		

Intrinsic Name	Result	Arguments	Families	Remarks
		Y: I*1, I*2, I*4, I*8, R*4, R*8, Z*8, Z*16		O
DCONJG	Z*16	Z: Z*16	G77, PGI, TRADITIONAL	E
DCOS	R*8	X: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DCOSD	R*8	X: R*8	PGI, TRADITIONAL	E
DCOSH	R*8	X: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DCOT	R*8	X: R*8	TRADITIONAL	E, P
DCOTAN	R*8	X: R*8	TRADITIONAL	E, P
DDIM	R*8	X: R*8 Y: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DERF		X: R*4, R*8	G77, PGI, TRADITIONAL	E, P
DERFC		X: R*4, R*8	G77, PGI, TRADITIONAL	E, P
DEXP	R*8	X: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DFLOAT	R*8	A: I*1, I*2, I*4, I*8	G77, PGI, TRADITIONAL	E
DFLOATI	R*8	A: I*2	TRADITIONAL	E
DFLOATJ	R*8	A: I*4	TRADITIONAL	E
DFLOATK	R*8	A: I*8	TRADITIONAL	E
DIGITS		X: I*1, I*2, I*4, I*8, R*4, R*8	ANSI, PGI, TRADITIONAL	E
DIM	R*4	X: R*4 Y: R*4	ANSI, G77, PGI, TRADITIONAL	E, P
DIM		X: R*8 Y: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DIM		X: I*1, I*2, I*4, I*8	ANSI, G77, PGI, TRADITIONAL	E, P

Intrinsic Name	Result	Arguments	Families	Remarks
		Y: I*1, I*2, I*4, I*8		
DIMAG	R*8	Z: Z*16	G77, PGI, TRADITIONAL	E
DINT	R*8	A: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DISABLE_IEEE_INTERRUPT	Subroutine	INTERRUPT: I*8	TRADITIONAL	E
DLOG	R*8	X: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DLOG10	R*8	X: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DMAX1			ANSI, G77, PGI, TRADITIONAL	See Std
DMIN1			ANSI, G77, PGI, TRADITIONAL	See Std
DMOD	R*8	A: R*8 P: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DNINT	R*8	A: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DOT_PRODUCT			ANSI, PGI, TRADITIONAL	See Std
DPROD	R*8	X: R*4, R*8 Y: R*4, R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DREAL	R*8	A: I*1, I*2, I*4, I*8, R*4, R*8, Z*8, Z*16	G77, PGI, TRADITIONAL	E
DSHIFTL		I: I*1, I*2, I*4, I*8 J: I*1, I*2, I*4, I*8 K: I*1, I*2, I*4, I*8	TRADITIONAL	E
DSHIFTR		I: I*1, I*2, I*4, I*8 J: I*1, I*2, I*4, I*8 K: I*1, I*2, I*4, I*8	TRADITIONAL	E
DSIGN	R*8	A: R*8 B: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DSIN	R*8	X: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DSIND	R*8	X: R*8	PGI, TRADITIONAL	E

Intrinsic Name	Result	Arguments	Families	Remarks
DSINH	R*8	X: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DSM_CHUNKSIZE	I*8	ARRAY: Any type, Array rank=any DIM: I*1, I*2, I*4, I*8	TRADITIONAL	
DSM_DISTRIBUTION_BLOCK	I*8	ARRAY: Any type, Array rank=any DIM: I*1, I*2, I*4, I*8	TRADITIONAL	
DSM_DISTRIBUTION_CYCLIC	I*8	ARRAY: Any type, Array rank=any DIM: I*1, I*2, I*4, I*8	TRADITIONAL	
DSM_DISTRIBUTION_STAR	I*8	ARRAY: Any type, Array rank=any DIM: I*1, I*2, I*4, I*8	TRADITIONAL	
DSM_ISDISTRIBUTED	I*8	ARRAY: Any type, Array rank=any	TRADITIONAL	
DSM_ISRESHAPED	I*8	ARRAY: Any type, Array rank=any	TRADITIONAL	
DSM_NUMCHUNKS	I*8	ARRAY: Any type, Array rank=any DIM: I*1, I*2, I*4, I*8	TRADITIONAL	
DSM_NUMTHREADS	I*8	ARRAY: Any type, Array rank=any DIM: I*1, I*2, I*4, I*8	TRADITIONAL	
DSM_REM_CHUNKSIZE	I*8	ARRAY: Any type, Array rank=any DIM: I*1, I*2, I*4, I*8 INDEX: I*1, I*2, I*4, I*8	TRADITIONAL	
DSM_THIS_CHUNKSIZE	I*8	ARRAY: Any type, Array rank=any DIM: I*1, I*2, I*4, I*8 INDEX: I*1, I*2, I*4, I*8	TRADITIONAL	
DSM_THIS_STARTINGINDEX	I*8	ARRAY: Any type, Array rank=any DIM: I*1, I*2, I*4, I*8 INDEX: I*1, I*2, I*4, I*8	TRADITIONAL	
DSM_THIS_THREADNUM	I*8	ARRAY: Any type, Array rank=any DIM: I*1, I*2, I*4, I*8 INDEX: I*1, I*2, I*4, I*8	TRADITIONAL	

Intrinsic Name	Result	Arguments	Families	Remarks
DSQRT	R*8	X: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DTAN	R*8	X: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DTAND	R*8	X: R*8	PGI, TRADITIONAL	E, P
DTANH	R*8	X: R*8	ANSI, G77, PGI, TRADITIONAL	E, P
DTIME	R*4	TARRAY: R*4, Array rank=1	G77, PGI, TRADITIONAL	
DTIME	Subroutine	TARRAY: R*4, Array rank=1 RESULT: R*4	G77, TRADITIONAL	
ENABLE_IEEE_INTERRUPT	Subroutine	INTERRUPT: I*8	TRADITIONAL	E
EOSHIFT			ANSI, PGI, TRADITIONAL	See Std
EPSILON		X: R*4, R*8	ANSI, PGI, TRADITIONAL	E
EQV		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 J: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8	PGI, TRADITIONAL	E
ERF		X: R*4, R*8	G77, PGI, TRADITIONAL	E, P
ERFC		X: R*4, R*8	G77, PGI, TRADITIONAL	E, P
ETIME	R*4	TARRAY: R*4, Array rank=1	G77, PGI, TRADITIONAL	
ETIME	Subroutine	TARRAY: R*4, Array rank=1 RESULT: R*4	G77, TRADITIONAL	
EXIT	Subroutine	STATUS: I*1, I*2, I*4, I*8	G77, PGI, TRADITIONAL	O
EXP	R*4	X: R*4, R*8, Z*8, Z*16	ANSI, G77, PGI, TRADITIONAL	E, P
EXPONENT		X: R*4, R*8	ANSI, PGI, TRADITIONAL	E

Intrinsic Name	Result	Arguments	Families	Remarks
FCD			TRADITIONAL	E
		I: I*1, I*2, I*4, I*8, CrayPtr		
		J: I*1, I*2, I*4, I*8		
FDATE	C		G77, PGI, TRADITIONAL	
FDATE	Subroutine		G77, PGI	
		DATE: C		
FETCH_AND_ADD			TRADITIONAL	E
		I: I*4		
		J: I*4		
FETCH_AND_ADD			TRADITIONAL	E
		I: I*8		
		J: I*8		
FETCH_AND_AND			TRADITIONAL	E
		I: I*4		
		J: I*4		
FETCH_AND_AND			TRADITIONAL	E
		I: I*8		
		J: I*8		
FETCH_AND_NAND			TRADITIONAL	E
		I: I*4		
		J: I*4		
FETCH_AND_NAND			TRADITIONAL	E
		I: I*8		
		J: I*8		
FETCH_AND_OR			TRADITIONAL	E
		I: I*4		
		J: I*4		
FETCH_AND_OR			TRADITIONAL	E
		I: I*8		
		J: I*8		
FETCH_AND_SUB			TRADITIONAL	E
		I: I*4		
		J: I*4		
FETCH_AND_SUB			TRADITIONAL	E
		I: I*8		
		J: I*8		
FETCH_AND_XOR			TRADITIONAL	E
		I: I*4		
		J: I*4		
FETCH_AND_XOR			TRADITIONAL	E
		I: I*8		
		J: I*8		
FGET	I*4		G77	
		C: C		
		STATUS: I*4		O
FGET	Subroutine		G77	
		C: C		
		STATUS: I*4		O
FGETC	I*4		G77, PGI	
		UNIT: I*4, I*8		
		C: C		
		STATUS: I*4		O
FGETC	Subroutine		G77	

Intrinsic Name	Result	Arguments	Families	Remarks
		UNIT: I*4, I*8		
		C: C		
		STATUS: I*4		O
FLOAT	R*4		ANSI, G77, PGI, TRADITIONAL	E
		A: I*1, I*2, I*4, I*8		
FLOATI	R*4		PGI, TRADITIONAL	E
		A: I*2		
FLOATJ	R*4		PGI, TRADITIONAL	E
		A: I*4		
FLOATK	R*4		PGI, TRADITIONAL	E
		A: I*8		
FLOOR			ANSI, PGI, TRADITIONAL	E
		A: R*4, R*8		
		KIND: I*1, I*2, I*4, I*8		O
FLUSH	Subroutine		G77, PGI	
		UNIT: I*4, I*8		O
		STATUS: I*4		O
FNUM	I*4		G77, TRADITIONAL	
		UNIT: I*4		
FPUT	I*4		G77	
		C: C		
		STATUS: I*4		O
FPUT	Subroutine		G77	
		C: C		
		STATUS: I*4		O
FPUTC	I*4		G77, PGI	
		UNIT: I*4, I*8		
		C: C		
		STATUS: I*4		O
FPUTC	Subroutine		G77	
		UNIT: I*4, I*8		
		C: C		
		STATUS: I*4		O
FP_CLASS	Depends on arg		TRADITIONAL	E
		X: R*4		
FP_CLASS	Depends on arg		TRADITIONAL	E
		X: R*4		
FP_CLASS	Depends on arg		TRADITIONAL	E
		X: R*8		
FP_CLASS	Depends on arg		TRADITIONAL	E
		X: R*8		
FRACTION			ANSI, PGI, TRADITIONAL	E
		X: R*4, R*8		
FREE	Subroutine		PGI, TRADITIONAL	E
		P: I*1, I*2, I*4, I*8, CrayPtr		
FSEEK	I*4		G77, PGI	

Intrinsic Name	Result	Arguments	Families	Remarks
		UNIT: I*4 OFFSET: I*4 WHENCE: I*4		
FSEEK	Subroutine		G77	
		UNIT: I*4 OFFSET: I*4 WHENCE: I*4		
FSEEK	Subroutine		G77	
		UNIT: I*4 OFFSET: I*8 WHENCE: I*4		
FSTAT	I*4		G77, PGI, TRADITIONAL	
		UNIT: I*1, I*2, I*4, I*8 SARRAY: I*1, I*2, I*4, I*8, Array rank=1 STATUS: I*1, I*2, I*4, I*8		O
FSTAT	Subroutine		G77	
		UNIT: I*1, I*2, I*4, I*8 SARRAY: I*1, I*2, I*4, I*8, Array rank=1 STATUS: I*1, I*2, I*4, I*8		O
FTELL	I*8		G77, PGI	
		UNIT: I*4		
FTELL	I*8		G77, PGI	
		UNIT: I*8		
FTELL	Subroutine		G77	
		UNIT: I*4 OFFSET: I*4		
FTELL	Subroutine		G77	
		UNIT: I*4 OFFSET: I*8		
FTELL	Subroutine		G77	
		UNIT: I*8 OFFSET: I*8		
GERROR	Subroutine		G77, PGI	
		MESSAGE: C		
GETARG	Subroutine		G77, PGI	
		POS: I*4 VALUE: C		
GETCWD	I*4		G77, PGI	
		NAME: C STATUS: I*4		O
GETCWD	Subroutine		G77	
		NAME: C STATUS: I*4		O
GETENV	Subroutine		G77, PGI	
		NAME: C VALUE: C		
GETGID	I*4		G77, PGI	
GETLOG	Subroutine		G77, PGI	
		LOGIN: C		
GETPID	I*4		G77, PGI	
GETUID	I*4		G77, PGI	

Intrinsic Name	Result	Arguments	Families	Remarks
GETPOS			TRADITIONAL	E
		I: I*1, I*2, I*4, I*8		
GET_COMMAND	Subroutine		ANSI, TRADITIONAL	
		COMMAND: C		O
		LENGTH: I*4		O
		STATUS: I*4		O
GET_COMMAND_ARGUMENT	Subroutine		ANSI, TRADITIONAL	
		NUMBER: I*4		
		VALUE: C		O
		LENGTH: I*4		O
		STATUS: I*4		O
GET_ENVIRONMENT_VARIABLE	Subroutine		ANSI, TRADITIONAL	
		NAME: C		
		VALUE: C		O
		LENGTH: I*4		O
		STATUS: I*4		O
		TRIM_NAME: L*4		O
GET_IEEE_EXCEPTIONS	Subroutine		TRADITIONAL	
		STATUS: I*8		
GET_IEEE_INTERRUPTS	Subroutine		TRADITIONAL	
		STATUS: I*8		
GET_IEEE_ROUNDING_MODE	Subroutine		TRADITIONAL	
		STATUS: I*8		
GET_IEEE_STATUS	Subroutine		TRADITIONAL	
		STATUS: I*8		
GMTIME	Subroutine		G77, PGI	
		STIME: I*4		
		TARRAY: I*4, Array rank=1		
HOSTNM	I*4		G77, PGI	
		NAME: C		
		STATUS: I*4		O
HOSTNM	Subroutine		G77	
		NAME: C		
		STATUS: I*4		O
HUGE			ANSI, PGI, TRADITIONAL	E
		X: I*1, I*2, I*4, I*8, R*4, R*8		
IABS	I*4		ANSI, G77, PGI, TRADITIONAL	E, P
		A: I*1, I*2, I*4, I*8		
IACHAR	I*4		ANSI, G77, PGI, TRADITIONAL	E
		C: C		
IAND	I*4		ANSI, G77, PGI, TRADITIONAL	E
		I: I*1, I*2, I*4, I*8		
		J: I*1, I*2, I*4, I*8		
IARGC	I*4		G77, PGI	
IBCHNG	I*4		TRADITIONAL	E
		I: I*1, I*2, I*4, I*8		
		POS: I*1, I*2, I*4, I*8		

Intrinsic Name	Result	Arguments	Families	Remarks
IBCLR	I*4	I: I*1, I*2, I*4, I*8 POS: I*1, I*2, I*4, I*8	ANSI, G77, PGI, TRADITIONAL	E
IBITS	I*4	I: I*1, I*2, I*4, I*8 POS: I*1, I*2, I*4, I*8 LEN: I*1, I*2, I*4, I*8	ANSI, G77, PGI, TRADITIONAL	E
IBSET	I*4	I: I*1, I*2, I*4, I*8 POS: I*1, I*2, I*4, I*8	ANSI, G77, PGI, TRADITIONAL	E
ICHAR	I*4	C: C	ANSI, G77, PGI, TRADITIONAL	E
IDATE	Subroutine	I: I*1 J: I*1 K: I*1	G77, PGI, TRADITIONAL	
IDATE	Subroutine	I: I*2 J: I*2 K: I*2	G77, PGI, TRADITIONAL	
IDATE	Subroutine	I: I*4 J: I*4 K: I*4	G77, PGI, TRADITIONAL	
IDATE	Subroutine	I: I*8 J: I*8 K: I*8	G77, PGI, TRADITIONAL	
IDATE	Subroutine	TARRAY: I*1, Array rank=1	G77, PGI, TRADITIONAL	
IDATE	Subroutine	TARRAY: I*2, Array rank=1	G77, PGI, TRADITIONAL	
IDATE	Subroutine	TARRAY: I*4, Array rank=1	G77, PGI, TRADITIONAL	
IDATE	Subroutine	TARRAY: I*8, Array rank=1	G77, PGI, TRADITIONAL	
IDIM	I*4	X: I*1, I*2, I*4, I*8 Y: I*1, I*2, I*4, I*8	ANSI, G77, PGI, TRADITIONAL	E, P
IDINT	I*4		ANSI, G77, PGI, TRADITIONAL	E

Intrinsic Name	Result	Arguments	Families	Remarks
		A: R*8		
IDNINT	I*4		ANSI, G77, PGI, TRADITIONAL	E, P
		A: R*8		
IEEE_BINARY_SCALE			TRADITIONAL	E
		Y: R*4, R*8		
		N: I*1, I*2, I*4, I*8		
IEEE_CLASS			TRADITIONAL	E
		X: R*4, R*8		
IEEE_COPY_SIGN			TRADITIONAL	E
		X: R*4, R*8		
		Y: R*4, R*8		
IEEE_EXPONENT			TRADITIONAL	E
		X: R*4, R*8		
		Y: I*1, I*2, I*4, I*8, R*4, R*8		O
IEEE_FINITE			TRADITIONAL	E
		X: R*4, R*8		
IEEE_INT			TRADITIONAL	E
		X: R*4, R*8		
		Y: I*1, I*2, I*4, I*8, R*4, R*8		O
IEEE_IS_NAN			TRADITIONAL	E
		X: R*4, R*8		
IEEE_NEXT_AFTER			TRADITIONAL	E
		X: R*4, R*8		
		Y: R*4, R*8		
IEEE_REAL			TRADITIONAL	E
		X: I*1, I*2, I*4, I*8, R*4, R*8		
		Y: R*4, R*8		O
IEEE_REMAINDER			TRADITIONAL	E
		X: R*4, R*8		
		Y: R*4, R*8		
IEEE_UNORDERED			TRADITIONAL	E
		X: R*4, R*8		
		Y: R*4, R*8		
IEOR	I*4		ANSI, G77, PGI, TRADITIONAL	E
		I: I*1, I*2, I*4, I*8		
		J: I*1, I*2, I*4, I*8		
IERRNO	I*4		G77, PGI	
IFIX	I*4		ANSI, G77, PGI, TRADITIONAL	E
		A: R*4, R*8		
IIABS	I*2		PGI, TRADITIONAL	E
		A: I*2		
IIAND	I*2		PGI, TRADITIONAL	E
		I: I*2		
		J: I*2		
IIBCHNG	I*2		TRADITIONAL	E
		I: I*2		
		POS: I*1, I*2, I*4, I*8		
IIBCLR	I*2		PGI, TRADITIONAL	E

Intrinsic Name	Result	Arguments	Families	Remarks
		I: I*2 POS: I*1, I*2, I*4, I*8		
IIBITS	I*2		PGI, TRADITIONAL	E
		I: I*2 POS: I*1, I*2, I*4, I*8 LEN: I*1, I*2, I*4, I*8		
IIBSET	I*2		PGI, TRADITIONAL	E
		I: I*2 POS: I*1, I*2, I*4, I*8		
IIDIM	I*2		PGI, TRADITIONAL	E
		X: I*2 Y: I*2		
IIDINT	I*2		PGI, TRADITIONAL	E
		A: R*8		
IIEOR	I*2		PGI, TRADITIONAL	E
		I: I*2 J: I*2		
IIFIX	I*2		PGI, TRADITIONAL	E
		A: R*4, R*8		
IINT	I*2		PGI, TRADITIONAL	E
		A: R*4		
IIOR	I*2		PGI, TRADITIONAL	E
		I: I*2 J: I*2		
IISHA	I*2		TRADITIONAL	E
		I: I*2 SHIFT: I*1, I*2, I*4, I*8		
IISHC	I*2		TRADITIONAL	E
		I: I*2 SHIFT: I*1, I*2, I*4, I*8		
IISHFT	I*2		PGI, TRADITIONAL	E
		I: I*2 SHIFT: I*1, I*2, I*4, I*8		
IISHFTC	I*2		PGI, TRADITIONAL	E
		I: I*2 SHIFT: I*1, I*2, I*4, I*8 SIZE: I*1, I*2, I*4, I*8		O
IISHL	I*2		TRADITIONAL	E
		I: I*2 SHIFT: I*1, I*2, I*4, I*8		
IISIGN	I*2		PGI, TRADITIONAL	E, P
		A: I*2 B: I*2		

Intrinsic Name	Result	Arguments	Families	Remarks
ILEN	Depends on arg	I: I*1	TRADITIONAL	E, P
ILEN	Depends on arg	I: I*2	TRADITIONAL	E, P
ILEN	Depends on arg	I: I*4	TRADITIONAL	E, P
ILEN	Depends on arg	I: I*8	TRADITIONAL	E, P
IMAG		Z: Z*8, Z*16	G77, TRADITIONAL	E
IMAGPART		Z: Z*8, Z*16	G77	E
IMOD	I*2	A: I*2 P: I*2	PGI, TRADITIONAL	E, P
IMVBITS	Subroutine	FROM: I*2 FROMPOS: I*1, I*2, I*4, I*8 LEN: I*1, I*2, I*4, I*8 TO: I*2 TOPOS: I*1, I*2, I*4, I*8	TRADITIONAL	E
INDEX	I*4	STRING: C SUBSTRING: C BACK: L*1, L*2, L*4, L*8	ANSI, G77, PGI, TRADITIONAL	E, P O
ININT	I*2	A: R*4, R*8	PGI, TRADITIONAL	E, P
INOT	I*2	I: I*2	PGI, TRADITIONAL	E
INT	I*4	A: I*1, I*2, I*4, I*8, R*4, R*8, Z*8, Z*16 KIND: I*1, I*2, I*4, I*8	ANSI, G77, PGI, TRADITIONAL	E O
INT2	I*2	A: I*1, I*2, I*4, I*8, R*4, R*8, Z*8, Z*16	G77, TRADITIONAL	E
INT4	I*4	A: I*1, I*2, I*4, I*8, R*4, R*8, Z*8, Z*16	TRADITIONAL	E
INT8	I*8	A: I*1, I*2, I*4, I*8, R*4, R*8, Z*8, Z*16	G77, PGI, TRADITIONAL	E
INT_MULT_UPPER		I: I*8 J: I*8		E
INT_MULT_UPPER				E

Intrinsic Name	Result	Arguments	Families	Remarks
		I:		
		J:		
IOR	I*4		ANSI, G77, PGI, TRADITIONAL	E
		I: I*1, I*2, I*4, I*8		
		J: I*1, I*2, I*4, I*8		
IRAND	I*4		G77, PGI	
		FLAG: I*4		O
IRTC	I*8		TRADITIONAL	
ISATTY	L*4		G77, PGI	
		UNIT: I*4		
ISHA			TRADITIONAL	E
		I: I*1, I*2, I*4, I*8		
		SHIFT: I*1, I*2, I*4, I*8		
ISHC			TRADITIONAL	E
		I: I*1, I*2, I*4, I*8		
		SHIFT: I*1, I*2, I*4, I*8		
ISHFT			ANSI, G77, PGI, TRADITIONAL	E
		I: I*1, I*2, I*4, I*8		
		SHIFT: I*1, I*2, I*4, I*8		
ISHFTC			ANSI, G77, PGI, TRADITIONAL	E
		I: I*1, I*2, I*4, I*8		
		SHIFT: I*1, I*2, I*4, I*8		
		SIZE: I*1, I*2, I*4, I*8		O
ISHL			TRADITIONAL	E
		I: I*1, I*2, I*4, I*8		
		SHIFT: I*1, I*2, I*4, I*8		
ISIGN	I*4		ANSI, G77, PGI, TRADITIONAL	E, P
		A: I*1, I*2, I*4, I*8		
		B: I*1, I*2, I*4, I*8		
ISNAN			TRADITIONAL	E
		X: R*4, R*8		
IS_IOSTAT_END	L*4		ANSI, TRADITIONAL	
		I: I*1, I*2, I*4, I*8		
IS_IOSTAT_EOR	L*4		ANSI, TRADITIONAL	
		I: I*1, I*2, I*4, I*8		
ITIME	Subroutine		G77, PGI	
		TARRAY: I*4, Array rank=1		
JDATE	C		TRADITIONAL	
JIABS	I*4		PGI, TRADITIONAL	E
		A: I*4		
JIAND	I*4		PGI, TRADITIONAL	E
		I: I*4		
		J: I*4		
JIBCHNG	I*4		TRADITIONAL	E
		I: I*4		
		POS: I*1, I*2, I*4, I*8		

Intrinsic Name	Result	Arguments	Families	Remarks
JIBCLR	I*4	I: I*4 POS: I*1, I*2, I*4, I*8	PGI, TRADITIONAL	E
JIBITS	I*4	I: I*4 POS: I*1, I*2, I*4, I*8 LEN: I*1, I*2, I*4, I*8	PGI, TRADITIONAL	E
JIBSET	I*4	I: I*4 POS: I*1, I*2, I*4, I*8	PGI, TRADITIONAL	E
JIDIM	I*4	X: I*4 Y: I*4	PGI, TRADITIONAL	E
JIDINT	I*4	A: R*8	PGI, TRADITIONAL	E
JIEOR	I*4	I: I*4 J: I*4	PGI, TRADITIONAL	E
JIFIX	I*4	A: R*4, R*8	PGI, TRADITIONAL	E
JINT	I*4	A: R*4	PGI, TRADITIONAL	E
JIOR	I*4	I: I*4 J: I*4	PGI, TRADITIONAL	E
JISHA	I*4	I: I*4 SHIFT: I*1, I*2, I*4, I*8	TRADITIONAL	E
JISHC	I*4	I: I*4 SHIFT: I*1, I*2, I*4, I*8	TRADITIONAL	E
JISHFT	I*4	I: I*4 SHIFT: I*1, I*2, I*4, I*8	PGI, TRADITIONAL	E
JISHFTC	I*4	I: I*4 SHIFT: I*1, I*2, I*4, I*8 SIZE: I*1, I*2, I*4, I*8	PGI, TRADITIONAL	E O
JISHL	I*4	I: I*4 SHIFT: I*1, I*2, I*4, I*8	TRADITIONAL	E
JISIGN	I*4		PGI, TRADITIONAL	E, P

Intrinsic Name	Result	Arguments	Families	Remarks
		A: I*4 B: I*4		
JMOD	I*4		PGI, TRADITIONAL	E, P
		A: I*4 P: I*4		
JMVBITS	Subroutine		TRADITIONAL	E
		FROM: I*4 FROMPOS: I*1, I*2, I*4, I*8 LEN: I*1, I*2, I*4, I*8 TO: I*4 TOPOS: I*1, I*2, I*4, I*8		
JNINT	I*4		TRADITIONAL	E, P
		A: R*4, R*8		
JNOT	I*4		PGI, TRADITIONAL	E
		I: I*4		
KIABS	I*8		PGI, TRADITIONAL	E
		A: I*8		
KIAND	I*8		PGI, TRADITIONAL	E
		I: I*8 J: I*8		
KIBCHNG	I*8		TRADITIONAL	E
		I: I*8 POS: I*1, I*2, I*4, I*8		
KIBCLR	I*8		PGI, TRADITIONAL	E
		I: I*8 POS: I*1, I*2, I*4, I*8		
KIBITS	I*8		PGI, TRADITIONAL	E
		I: I*8 POS: I*1, I*2, I*4, I*8 LEN: I*1, I*2, I*4, I*8		
KIBSET	I*8		PGI, TRADITIONAL	E
		I: I*8 POS: I*1, I*2, I*4, I*8		
KIDIM	I*8		PGI, TRADITIONAL	E
		X: I*8 Y: I*8		
KIDINT	I*8		TRADITIONAL	E
		A: R*8		
KIEOR	I*8		TRADITIONAL	E
		I: I*8 J: I*8		
KIFIX	I*8		PGI, TRADITIONAL	E
		A: R*4, R*8		
KILL	I*4		G77, PGI, TRADITIONAL	

Intrinsic Name	Result	Arguments	Families	Remarks
		PID: I*4 SIG: I*4		
KILL	Subroutine		G77, TRADITIONAL	
		PID: I*4 SIG: I*4 STATUS: I*4		O
KIND	I*4		ANSI, PGI, TRADITIONAL	E
		X: Any type		
KINT	I*8		TRADITIONAL	E
		A: R*4		
KIOR	I*8		PGI, TRADITIONAL	E
		I: I*8 J: I*8		
KISHA	I*8		TRADITIONAL	E
		I: I*8 SHIFT: I*1, I*2, I*4, I*8		
KISHC	I*8		TRADITIONAL	E
		I: I*8 SHIFT: I*1, I*2, I*4, I*8		
KISHFT	I*8		PGI, TRADITIONAL	E
		I: I*8 SHIFT: I*1, I*2, I*4, I*8		
KISHFTC	I*8		PGI, TRADITIONAL	E
		I: I*8 SHIFT: I*1, I*2, I*4, I*8 SIZE: I*1, I*2, I*4, I*8		O
KISHL	I*8		TRADITIONAL	E
		I: I*8 SHIFT: I*1, I*2, I*4, I*8		
KISIGN	I*8		PGI, TRADITIONAL	E, P
		A: I*8 B: I*8		
KMOD	I*8		PGI, TRADITIONAL	E, P
		A: I*8 P: I*8		
KMVBITS	Subroutine		TRADITIONAL	E
		FROM: I*8 FROMPOS: I*1, I*2, I*4, I*8 LEN: I*1, I*2, I*4, I*8 TO: I*8 TOPOS: I*1, I*2, I*4, I*8		
KNINT	I*8		PGI, TRADITIONAL	E, P
		A: R*4, R*8		
KNOT	I*8		PGI, TRADITIONAL	E
		I: I*8		

Intrinsic Name	Result	Arguments	Families	Remarks
LBOUND			ANSI, PGI, TRADITIONAL	See Std
LEN	I*4		ANSI, G77, PGI, TRADITIONAL	E, P
		STRING: C		
LENGTH			TRADITIONAL	E
		I: I*1, I*2, I*4, I*8		
LEN_TRIM	I*4		ANSI, G77, PGI, TRADITIONAL	E
		STRING: C		
LGE	C		ANSI, G77, PGI, TRADITIONAL	E
		STRING_A: C		
		STRING_B: C		
LGT	C		ANSI, G77, PGI, TRADITIONAL	E
		STRING_A: C		
		STRING_B: C		
LINK	I*4		G77, PGI	
		PATH1: C		
		PATH2: C		
LINK	Subroutine		G77	
		PATH1: C		
		PATH2: C		
		STATUS: I*4		O
LLE	C		ANSI, G77, PGI, TRADITIONAL	E
		STRING_A: C		
		STRING_B: C		
LLT	C		ANSI, G77, PGI, TRADITIONAL	E
		STRING_A: C		
		STRING_B: C		
LNBLNK	I*4		G77, PGI	
		STRING: C		
LOC	I*8		G77, PGI, TRADITIONAL	
		I:Any type, Array rank=any		
LOCK_RELEASE	Subroutine		TRADITIONAL	E
		I: I*4, I*8		O
LOCK_TEST_AND_SET			TRADITIONAL	E
		I: I*4		
		J: I*4		
LOCK_TEST_AND_SET			TRADITIONAL	E
		I: I*8		
		J: I*8		
LOG	R*4		ANSI, G77, PGI, TRADITIONAL	E
		X: R*4, R*8, Z*8, Z*16		
LOG10	R*4		ANSI, G77, PGI, TRADITIONAL	E
		X: R*4, R*8		
LOG2_IMAGES	I*4		TRADITIONAL	

Intrinsic Name	Result	Arguments	Families	Remarks
LOGICAL	L*4		ANSI, PGI, TRADITIONAL	E
		L: L*1, L*2, L*4, L*8		
		KIND: I*1, I*2, I*4, I*8		O
LONG	I*4		G77, TRADITIONAL	E
		A: I*1, I*2, I*4, I*8, R*4, R*8, Z*8, Z*16		
LSHIFT			G77, PGI, TRADITIONAL	E
		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8		
		POSITIVE_SHIFT: I*1, I*2, I*4, I*8		
LSTAT	I*4		G77, PGI	
		FILE: C		
		SARRAY: I*4, Array rank=1		
		STATUS: I*4		O
LSTAT	Subroutine		G77	
		FILE: C		
		SARRAY: I*4, Array rank=1		
		STATUS: I*4		O
LTIME	Subroutine		G77, PGI	
		STIME: I*4		
		TARRAY: I*4, Array rank=1		
MALLOC			PGI, TRADITIONAL	E
		I: I*1, I*2, I*4, I*8		
MASK			TRADITIONAL	E
		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8		
MATMUL			ANSI, PGI, TRADITIONAL	See Std
MAX			ANSI, G77, PGI, TRADITIONAL	See Std
MAX0			ANSI, G77, PGI, TRADITIONAL	See Std
MAX1			ANSI, G77, PGI, TRADITIONAL	See Std
MAXEXPONENT			ANSI, PGI, TRADITIONAL	E
		X: R*4, R*8		
MAXLOC			ANSI, PGI, TRADITIONAL	See Std
MAXVAL			ANSI, PGI, TRADITIONAL	See Std
MCLOCK	I*4		G77, PGI	
MCLOCK8	I*8		G77	
MEMORY_BARRIER	Subroutine		TRADITIONAL	E
MERGE			ANSI, PGI, TRADITIONAL	E
		TSOURCE: Any type		
		FSOURCE: Any type		
		MASK: L*1, L*2, L*4, L*8		

Intrinsic Name	Result	Arguments	Families	Remarks
MIN			ANSI, G77, PGI, TRADITIONAL	See Std
MIN0			ANSI, G77, PGI, TRADITIONAL	See Std
MIN1			ANSI, G77, PGI, TRADITIONAL	See Std
MINEXPONENT			ANSI, PGI, TRADITIONAL	E
		X: R*4, R*8		
MINLOC			ANSI, PGI, TRADITIONAL	See Std
MINVAL			ANSI, PGI, TRADITIONAL	See Std
MOD	I*4		ANSI, G77, PGI, TRADITIONAL	E, P
		A: I*1, I*2, I*4, I*8, R*4, R*8		
		P: I*1, I*2, I*4, I*8, R*4, R*8		
MODULO			ANSI, PGI, TRADITIONAL	E
		A: I*1, I*2, I*4, I*8, R*4, R*8		
		P: I*1, I*2, I*4, I*8, R*4, R*8		
MVBITS	Subroutine		ANSI, G77, PGI, TRADITIONAL	E
		FROM: I*1, I*2, I*4, I*8		
		FROMPOS: I*1, I*2, I*4, I*8		
		LEN: I*1, I*2, I*4, I*8		
		TO: I*1, I*2, I*4, I*8		
		TOPOS: I*1, I*2, I*4, I*8		
NAND_AND_FETCH			TRADITIONAL	E
		I: I*4		
		J: I*4		
NAND_AND_FETCH			TRADITIONAL	E
		I: I*8		
		J: I*8		
NEAREST			ANSI, PGI, TRADITIONAL	E
		X: R*4, R*8		
		S: R*4, R*8		
NEQV			PGI, TRADITIONAL	E
		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8		
		J: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8		
NINT	I*4		ANSI, G77, PGI, TRADITIONAL	E, P
		A: R*4, R*8		
		KIND: I*1, I*2, I*4, I*8		O
NOT			ANSI, G77, PGI, TRADITIONAL	E
		I: I*1, I*2, I*4, I*8		
NULL			ANSI, PGI, TRADITIONAL	

Intrinsic Name	Result	Arguments	Families	Remarks
		MOLD: Any type, Array rank=any		
NUM_IMAGES	I*4		TRADITIONAL	
OMP_DESTROY_LOCK	Subroutine	LOCK: I*4, I*8	OMP	
OMP_DESTROY_NEST_LOCK	Subroutine	LOCK: I*4, I*8	OMP	
OMP_GET_DYNAMIC	Depends on arg		OMP	
OMP_GET_MAX_THREADS	Depends on arg		OMP	
OMP_GET_NESTED	Depends on arg		OMP	
OMP_GET_NUM_PROCS	Depends on arg		OMP	
OMP_GET_NUM_THREADS	Depends on arg		OMP	
OMP_GET_THREAD_NUM	Depends on arg		OMP	
OMP_GET_WTICK	R*8		OMP	
OMP_GET_WTIME	R*8		OMP	
OMP_INIT_LOCK	Subroutine	LOCK: I*4, I*8	OMP	
OMP_INIT_NEST_LOCK	Subroutine	LOCK: I*4, I*8	OMP	
OMP_IN_PARALLEL	Depends on arg		OMP	
OMP_SET_DYNAMIC	Subroutine	DYNAMIC_THREADS: L*4, L*8	OMP	
OMP_SET_LOCK	Subroutine	LOCK: I*4, I*8	OMP	
OMP_SET_NESTED	Subroutine	NESTED: L*4, L*8	OMP	
OMP_SET_NEST_LOCK	Subroutine	LOCK: I*4, I*8	OMP	
OMP_SET_NUM_THREADS	Subroutine	NUM_THREADS: I*4, I*8	OMP	
OMP_TEST_LOCK	Depends on arg	LOCK: I*4, I*8	OMP	
OMP_TEST_NEST_LOCK	Depends on arg	LOCK: I*4, I*8	OMP	
OMP_UNSET_LOCK	Subroutine	LOCK: I*4, I*8	OMP	
OMP_UNSET_NEST_LOCK	Subroutine	LOCK: I*4, I*8	OMP	
OR		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 J: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8	G77, PGI, TRADITIONAL	E
OR_AND_FETCH		I: I*4 J: I*4	TRADITIONAL	E
OR_AND_FETCH		I: I*8 J: I*8	TRADITIONAL	E
PACK			ANSI, PGI, TRADITIONAL	See Std
PERROR	Subroutine		G77, PGI	

Intrinsic Name	Result	Arguments	Families	Remarks
		STRING: C		
POPCNT			TRADITIONAL	E
		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8		
POPPAR			TRADITIONAL	E
		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8		
PRECISION			ANSI, PGI, TRADITIONAL	E
		X: R*4, R*8, Z*8, Z*16		
PRESENT			ANSI, PGI, TRADITIONAL	E
		A: Procedure, Any type		
PRESENT			ANSI, PGI, TRADITIONAL	E
		A: Any type		
PRODUCT			ANSI, PGI, TRADITIONAL	See Std
RADIX			ANSI, PGI, TRADITIONAL	E
		X: I*1, I*2, I*4, I*8, R*4, R*8		
RAND	R*8		G77, PGI	
		FLAG: I*4		O
RANDOM_NUMBER	Subroutine		ANSI, PGI, TRADITIONAL	E
		HARVEST: R*4, R*8		O
RANDOM_SEED	Subroutine		ANSI, PGI, TRADITIONAL	
		SIZE: I*1, I*2, I*4, I*8		O
		PUT: I*1, I*2, I*4, I*8, Array rank=1		O
		GET: I*1, I*2, I*4, I*8, Array rank=1		O
RANF			TRADITIONAL	E
RANGE			ANSI, PGI, TRADITIONAL	E
		X: I*1, I*2, I*4, I*8, R*4, R*8, Z*8, Z*16		
REAL	R*4		ANSI, G77, PGI, TRADITIONAL	E
		A: I*1, I*2, I*4, I*8, R*4, R*8, Z*8, Z*16		
		KIND: I*1, I*2, I*4, I*8		O
REALPART	R*4		G77	E
		A: I*1, I*2, I*4, I*8, R*4, R*8, Z*8, Z*16		
		KIND: I*1, I*2, I*4, I*8		O
REMOTE_WRITE_BARRIER	Subroutine		TRADITIONAL	E
REM_IMAGES	I*4		TRADITIONAL	
RENAME	I*4		G77, PGI	
		PATH1: C		
		PATH2: C		
		STATUS: I*4		O
RENAME	Subroutine		G77	

Intrinsic Name	Result	Arguments	Families	Remarks
		PATH1: C		
		PATH2: C		
		STATUS: I*4		O
REPEAT	Depends on arg		ANSI, PGI, TRADITIONAL	
		STRING: C		
		NCOPIES: I*1, I*2, I*4, I*8		
RESHAPE			ANSI, PGI, TRADITIONAL	See Std
RRSPACING			ANSI, PGI, TRADITIONAL	E
		X: R*4, R*8		
RSHIFT			G77, PGI, TRADITIONAL	E
		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8		
		NEGATIVE_SHIFT: I*1, I*2, I*4, I*8		
RTC			TRADITIONAL	E
SCALE			ANSI, PGI, TRADITIONAL	E
		X: R*4, R*8		
		I: I*1, I*2, I*4, I*8		
SCAN	I*4		ANSI, PGI, TRADITIONAL	E
		STRING: C		
		SET: C		
		BACK: L*1, L*2, L*4, L*8		O
SECNDS	R*4		G77, PGI	
		T: R*4		
SECOND	R*4		G77	
		SECONDS: R*4		O
SECOND	Subroutine		G77	
		SECONDS: R*4		
SELECTED_INT_KIND			ANSI, PGI, TRADITIONAL	
		R: I*1, I*2, I*4, I*8		
SELECTED_REAL_KIND	Depends on arg		ANSI, PGI, TRADITIONAL	
		P: I*1, I*2, I*4, I*8		O
		R: I*1, I*2, I*4, I*8		O
SET_EXPONENT			ANSI, PGI, TRADITIONAL	E
		X: R*4, R*8		
		I: I*1, I*2, I*4, I*8		
SET_IEEE_EXCEPTION	Subroutine		TRADITIONAL	E
		EXCEPTION: I*8		
SET_IEEE_EXCEPTIONS	Subroutine		TRADITIONAL	
		STATUS: I*8		
SET_IEEE_INTERRUPTS	Subroutine		TRADITIONAL	
		STATUS: I*8		
SET_IEEE_ROUNDING_MODE	Subroutine		TRADITIONAL	
		STATUS: I*8		
SET_IEEE_STATUS	Subroutine		TRADITIONAL	

Intrinsic Name	Result	Arguments	Families	Remarks
		STATUS: I*8		
SHAPE			ANSI, PGI, TRADITIONAL	See Std
SHIFT			PGI, TRADITIONAL	E
		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 J: I*1, I*2, I*4, I*8		
SHIFTA			TRADITIONAL	E
		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 J: I*1, I*2, I*4, I*8		
SHIFTL			TRADITIONAL	E
		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 J: I*1, I*2, I*4, I*8		
SHIFTR			TRADITIONAL	E
		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8 J: I*1, I*2, I*4, I*8		
SHORT	I*2		G77, TRADITIONAL	E
		A: I*1, I*2, I*4, I*8, R*4, R*8, Z*8, Z*16		
SIGN	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		A: I*1, I*2, I*4, I*8, R*4, R*8 B: I*1, I*2, I*4, I*8, R*4, R*8		
SIGNAL	I*8		G77, PGI, TRADITIONAL	
		NUMBER: I*1, I*2, I*4, I*8 HANDLER: Procedure IGNDFL: I*4		O
SIGNAL	I*8		G77, PGI, TRADITIONAL	
		NUMBER: I*1, I*2, I*4, I*8 HANDLER: I*4		
SIGNAL	I*8		G77, PGI, TRADITIONAL	
		NUMBER: I*1, I*2, I*4, I*8 HANDLER: I*8		
SIGNAL	Subroutine		G77, PGI, TRADITIONAL	
SIN	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		X: R*4, R*8, Z*8, Z*16		
SIND	R*4		PGI, TRADITIONAL	E, P
		X: R*4, R*8		
SINH	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		X: R*4, R*8		
SIZE			ANSI, PGI, TRADITIONAL	See Std

Intrinsic Name	Result	Arguments	Families	Remarks
SIZEOF	I*8	X: Any type, Array rank=any	TRADITIONAL	
SLEEP	Subroutine	SECONDS: I*4	G77, PGI	
SNGL	R*4	A: R*8	ANSI, G77, PGI, TRADITIONAL	E
SPACING		X: R*4, R*8	ANSI, PGI, TRADITIONAL	E
SPREAD			ANSI, PGI, TRADITIONAL	See Std
SQRT	R*4	X: R*4, R*8, Z*8, Z*16	ANSI, G77, PGI, TRADITIONAL	E, P
SRAND	Subroutine	SEED: I*4	G77, PGI	
STAT	I*4	FILE: C SARRAY: I*4, Array rank=1 STATUS: I*4	G77, PGI, TRADITIONAL	O
STAT	Subroutine	FILE: C SARRAY: I*4, Array rank=1 STATUS: I*4	G77, TRADITIONAL	O
SUB_AND_FETCH		I: I*4 J: I*4	TRADITIONAL	E
SUB_AND_FETCH		I: I*8 J: I*8	TRADITIONAL	E
SUM			ANSI, PGI, TRADITIONAL	See Std
SYMLNK	I*4	PATH1: C PATH2: C STATUS: I*4	G77, PGI	O
SYMLNK	Subroutine	PATH1: C PATH2: C STATUS: I*4	G77	O
SYNCHRONIZE	Subroutine		TRADITIONAL	E
SYNC_IMAGES	Subroutine		TRADITIONAL	
SYNC_IMAGES	Subroutine	IMAGE: I*1, I*2, I*4, I*8	TRADITIONAL	
SYNC_IMAGES	Subroutine	IMAGE: I*1, I*2, I*4, I*8, Array rank=1	TRADITIONAL	
SYSTEM	I*4	COMMAND: C STATUS: I*4	G77, PGI	O
SYSTEM	Subroutine		G77	

Intrinsic Name	Result	Arguments	Families	Remarks
		COMMAND: C		
		STATUS: I*4		O
SYSTEM_CLOCK	Subroutine		ANSI, G77, PGI, TRADITIONAL	
		COUNT: I*4		O
		COUNT_RATE: I*4		O
		COUNT_MAX: I*4		O
SYSTEM_CLOCK	Subroutine		ANSI, G77, PGI, TRADITIONAL	
		COUNT: I*8		O
		COUNT_RATE: I*8		O
		COUNT_MAX: I*8		O
TAN	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		X: R*4, R*8		
TAND	R*4		PGI, TRADITIONAL	E
		X: R*4, R*8		
TANH	R*4		ANSI, G77, PGI, TRADITIONAL	E, P
		X: R*4, R*8		
TEST_IEEE_EXCEPTION			TRADITIONAL	E
		EXCEPTION: I*8		
TEST_IEEE_INTERRUPT			TRADITIONAL	E
		INTERRUPT: I*8		
THIS_IMAGE	Depends on arg		TRADITIONAL	
		ARRAY: Any type, Array rank=any		
		DIM: I*1, I*2, I*4, I*8		O
TIME	I*4		G77, PGI, TRADITIONAL	
TIME8	I*8		G77, TRADITIONAL	
TIME	Subroutine		G77	
		BUF: C		
TINY			ANSI, PGI, TRADITIONAL	E
		X: R*4, R*8		
TRANSFER			ANSI, PGI, TRADITIONAL	See Std
TRANSPOSE	Depends on arg		ANSI, PGI, TRADITIONAL	
		MATRIX: Any type, Array rank=2		
TRIM	Depends on arg		ANSI, PGI, TRADITIONAL	
		STRING: C		
TTYNAM	C		G77, PGI	
		UNIT: I*4		
TTYNAM	Subroutine		G77	
		UNIT: I*4		
		NAME: C		
UBOUND			ANSI, PGI, TRADITIONAL	See Std

Intrinsic Name	Result	Arguments	Families	Remarks
UMASK	I*4		G77	
		MASK: I*4		
UMASK	Subroutine		G77	
		MASK: I*4		
		OLD: I*4		O
UNIT			TRADITIONAL	E
		I: I*1, I*2, I*4, I*8		
UNLINK	I*4		G77, PGI	
		FILE: C		
		STATUS: I*4		O
UNLINK	Subroutine		G77	
		FILE: C		
		STATUS: I*4		O
UNPACK			ANSI, PGI, TRADITIONAL	See Std
VERIFY	I*4		ANSI, PGI, TRADITIONAL	E
		STRING: C		
		SET: C		
		BACK: L*1, L*2, L*4, L*8		O
WRITE_MEMORY_BARRIER	Subroutine		TRADITIONAL	E
XOR			G77, PGI, TRADITIONAL	E
		I: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8		
		J: I*1, I*2, I*4, I*8, R*4, R*8, CrayPtr, L*1, L*2, L*4, L*8		
XOR_AND_FETCH			TRADITIONAL	E
		I: I*4		
		J: I*4		
XOR_AND_FETCH			TRADITIONAL	E
		I: I*8		
		J: I*8		
ZABS	R*8		G77, TRADITIONAL	E, P
		A: Z*16		
ZCOS	Z*16		G77, TRADITIONAL	E, P
		X: Z*16		
ZEXP	Z*16		G77, TRADITIONAL	E, P
		X: Z*16		
ZLOG	Z*16		G77, TRADITIONAL	E, P
		X: Z*16		
ZSIN	Z*16		G77, TRADITIONAL	E, P
		X: Z*16		
ZSQRT	Z*16		G77, TRADITIONAL	E, P
		X: Z*16		

C.4 Fortran Intrinsic extensions

Standard Fortran intrinsic procedures are documented in ISO 1539-1 or any good textbook on Fortran 95. This section documents procedures that are extensions to the standard, referring to argument names shown in the table of intrinsics in Section C.3.

abort
Prints a message and then, like the C library function `abort`, stops the program.

access
Like the C library function `access`, returns zero if the file named by `name` satisfies the requirements indicated by `mode`, but otherwise returns the error code from the C library value `errno`.

Trailing blanks in `name` are ignored (you can prevent this by using `char(0)` to place a null character after the last significant character.)

`mode` may contain any of the following:

`r` Readable

`w` Writable

`x` Executable

`' '` File exists

alarm
Uses the C library functions `alarm` and `signal` to wait the time indicated by `seconds` and then execute the external subroutine `handler`. `status` returns the number of seconds remaining until the previously scheduled alarm would have taken place, or `0` if no alarm was pending.

and
Bitwise boolean `AND`

besj0, besj1, besjn, besy0, besy1, besyn
Fortran interfaces to C library functions `j0`, `j1`, `jn`, `y0`, `y1`, and `yn` (Bessel functions.)

cdabs, cdcos, cdexp, cdlog, cdsin, cdsqrt
Specific names for various mathematical functions having an argument of type `complex*16`.

chdir
Like the C library function `chdir`, sets the current working directory to `dir`. The function form returns `0` on success, but otherwise returns the error code from the C library value `errno`. The subroutine form sets `status` to the value that the function form would return.

Trailing blanks in `dir` are ignored (you can prevent this by using `char(0)` to place a null character after the last significant character.)

chmod
Like the POSIX command `chmod`, changes the access permissions of file `name` according to `mode`. See the operating system documentation for the characters allowed in `mode`. The function form returns `0` on success, but

otherwise returns the error code from the C library value `errno`. The subroutine form sets `status` to the value which the function form would return.

Trailing blanks in `name` are ignored (you can prevent this by using `char(0)` to place a null character after the last significant character.)

ctime
Like the C library function `ctime`, converts `stime` (which can be obtained from the intrinsic `time8`) to a string of the form `Thu Mar 2 12:45:36 PST 2006`. The function form returns that string. The subroutine form sets `result` to that string.

date
Set the `date` argument to a string of the form `16-Mar-06` (`DD-MMM-YY`).

dbesj0, dbesj1, dbesjn, dbesy0, dbesy1, dbesyn
Fortran interfaces to C library functions `j0l`, `j1l`, `jnl`, `y0l`, `y1l`, and `ynl` (Bessel functions.)

dcmplx
Specific name for a function that converts its argument to type `complex*16.`

dconj
Specific name for complex conjugate whose argument is type `complex*16.`

derf, derfc
Fortran interface to C library function in `erf` and `erfc` (3m)

dfloat
Specific name for function that converts its argument to type `real*8.`

dimag
Specific name for a function that returns the imaginary part of a `complex*16` argument.

dreal
Specific name for a function that converts its argument to type `real*8.`

dtime
Find out the number of seconds of CPU time consumed by this process since the previous call to `dtime` (or, if there was no previous call, since the start of execution). `tarray(1)` gives user CPU time and `tarray(2)` gives system CPU time. The function form returns the sum of those times. The subroutine form sets `result` to the sum of those times.

erf, erfc
Fortran interface to C library functions described in `erff` and `erfcf` (3m)

etime
Find out the number of seconds of CPU time consumed by this process since the start of execution. `tarray(1)` gives user CPU time and `tarray(2)` gives system CPU time. The function form returns the sum of those times. The subroutine form sets `result` to the sum of those times.

exit
Like the C library function `exit`, terminate the process and return the value `status` to the process (usually the shell) that caused this process to execute. `status` defaults to `0`. Open Fortran logical units are flushed and closed.

fdate
The subroutine form is equivalent to `call ctime(date, time8())`. The function form is equivalent to `ctime(time8())`.

fget
Like `fgetc`, but uses logical unit 5.

fgetc
Fortran interface to the C library function `fgetc`. Reads into `c` a single character from logical unit `unit`, treating that unit as if it were a stream of bytes. The function form returns `0` for success, `-1` for end-of-file, or an error code from the C library value `errno`. The subroutine sets `status` to the value that the function would return.

Between the opening and closing of a file, you should use either stream intrinsics (`fget`, `fgetc`, `fput`, `fputc`, `fseek`, and `ftell`) or standard Fortran I/O, but not both.

flush
Flush buffered I/O for logical unit `unit`. If `unit` is omitted, flush all logical units.

fnum
Return the POSIX file descriptor corresponding to the open Fortran logical unit `unit`.

fput
Like `fputc`, but uses logical unit 6.

fputc
Fortran interface to the C library function `fput`. Writes to logical unit `unit` a single character `c`, treating that unit as if it were a stream of bytes. The function form returns `0` for success, `-1` for end-of-file, or an error code from the C library value `errno`. The subroutine sets `status` to the value that the function would return.

Between the opening and closing of a file, you should use either stream intrinsics (`fget`, `fgetc`, `fput`, `fputc`, `fseek`, and `ftell`) or standard Fortran I/O, but not both.

fseek
Fortran interface to the C library function `fseek`, which treats logical unit `unit` as a stream of bytes, and changes to `offset` the position pointer used by the next stream intrinsicm which reads or writes the file. If `whence` is `0`, `offset` counts bytes from the beginning of the file; if `whence` is `1`, `offset` positions the pointer relative to the current position; and if `whence` is `2`, `offset` positions the pointer relative to the end of the file. The function form returns `0` on success, or an error code from the C library value `errno`.

Between the opening and closing of a file, you should use either stream intrinsics (`fget`, `fgetc`, `fput`, `fputc`, `fseek`, and `ftell`) or standard Fortran I/O, but not both.

fstat
Fortran interface to the C library function `fstat`. Stores in `sarray` information about the file opened on logical unit `unit`. The function form returns `0` on success, or an error code from the C library variable `errno`. The subroutine form sets `status` to the value which the function would return.

`sarray` must have thirteen elements:
1. ID of device containing file
2. Inode number
3. File mode
4. Number of links
5. UID of owner
6. GID of owner
7. ID of device containing directory entry for file
8. Size of file in bytes
9. Time of last access
10. Time of last modification
11. Time of last file status change
12. Preferred I/O block size (`-1` if not available)
13. Number of blocks allocated (`-1` if not available)

Except for elements 12 and 13, values are set to `0` if they are not available from the relevant file system.

ftell
Fortran interface to the C library function `ftell`. Treats logical unit `unit` as a stream of bytes. The function form returns the offset from the beginning of the file to the position pointer used to read or write the file, or `-1` to indicate an error. The subroutine form sets `offset` to the value which the function would return.

gerror
Fortran interface to the C library function `strerror`. Sets `message` to the error message corresponding to the error code from the C library variable `errno`.

getarg
Stores into `value` an argument from the command line used to execute this process. `pos` is an index into the argument list (where `0` identifies the name of the program, `1` identifies the first argument, etc.) Intrinsic `iargc` provides the number of arguments available.

getcwd
Fortran interface to the C library function `getcwd`. Sets `name` to the current working directory name. The function form returns `0` for success, or an error code from the C library value `errno`. The subroutine form sets `status` to the value which the function would return.

getenv
Fortran interface to the C library function `getenv`. Sets `value` to the value of environment variable whose name is `name`, or to blanks if the variable is missing or not set.

Trailing blanks in `name` are ignored (you can prevent this by using `char(0)` to place a null character after the last significant character).

getgid
Like the POSIX function `getgid`, returns the group ID for this process.

getlog
Sets `login` to the login name for this process.

getpid
Like the POSIX function `getpid`, returns the process ID for this process.

getuid
Like the POSIX function `getuid`, returns the process ID for this process.

gmtime
Fortran interface to the C library function `gmtime`. Sets `tarray` to the broken-down time corresponding to `stime`, which can be obtained from the intrinsic `time8`. All values are in Coordinated Universal Time.

`tarray` must have nine elements:
1. Seconds since the last minute, ranging `0..61` (due to leap seconds)
2. Minutes since the last hour, ranging `0..59`
3. Hours since midnight, ranging `0..23`
4. Day of month, ranging `0..31`
5. Month, ranging `0..11`
6. Years since 1900
7. Days since Sunday, ranging `0..6`
8. Days since January 1, ranging `0..365`
9. Positive if daylight savings time is in effect, zero if not, or negative if unknown

hostnm
Fortran interface to the POSIX function `gethostname`. Sets `name` to the network name of the host computer. The function form returns `0` on success, or an error code from the C library value `errno`. The subroutine form sets `status` to the value that the function would return.

iargc
Return the number of arguments on the command line used to execute this program, not including the program name itself.

idate
The single-argument version stores in `tarray`, which must have three elements, the current local date:
1. Day, ranging `1..31`
2. Month, ranging `1..12`
3. Year, using 4 digits

The three-argument version sets its arguments to the month, day, and year. Note that the order is different from that of the one-argument version.

ierrno
Returns the C library value `errno`, which is the last error code set by a C library (or Linux system) function. Note that a function which does not encounter an error may not set this value back to zero.

imag
Return the imaginary part of a complex number without altering precision.

imagpart
Imaginary part of a complex number (synonym for standard intrinsic `aimag`, which in Fortran 95 preserves the precision of its argument).

int2
Convert to type `integer*2`.

int4
Convert to type `integer*4`.

int8
Convert to type `integer*8`.

irand
Fortran interface to POSIX function `rand`. Returns a uniform pseudorandom integer. If `flag` is `0`, return the next number in the current sequence; if `flag` is `1`, call POSIX function `srand(0)`; otherwise call `srand(flag)` to seed a new sequence.

isatty
Fortran interface to Linux function `isatty`. Returns `.true.` if logical unit `unit` is associated with an interactive terminal device.

itime
Store in `tarray`, which must have three elements, the current local time:
1. Hour, ranging `0..23`
2. Minutes, ranging `0..59`
3. Seconds, ranging `0..60` (to allow for leap seconds)

kill
Fortran interface to the POSIX function `kill`. Send to the process whose ID is `pid` the signal whose number is `signal`. The function form returns `0` on success, or an error code from the C library value `errno`. The subroutine form sets `status` to the value which the function would return.

link
Fortran interface to the POSIX function `link`. Creates a hard link `path2` pointing to the same file as `path1`. The function form returns `0` on success, or an error code from the C library value `errno`. The subroutine form sets `status` to the value which the function would return.

Trailing blanks in `path1` and `path2` are ignored (you can prevent this by using `char(0)` to place a null character after the last significant character.)

lnblnk
Returns the length of its argument, neglecting trailing blanks (synonym for standard function `len_trim`.)

loc
Returns address of argument im memory.

long
Convert to type `integer*4`.

lshift
Bitwise left shift. High-order bit is not treated as a sign bit. Shift count must be nonnegative and less than the bit-size of the data.

lstat
Fortran interface to the POSIX function `lstat`. Store in array `sarray` information about the file named `file`; if that is a symbolic link, describe the link rather than the target of the link (cf. `stat`). The function form returns `0`, or an error code from the C library value `errno`.

Trailing blanks in `file` are ignored (you can prevent this by using `char(0)` to place a null character after the last significant character).

`sarray` must have thirteen elements:
1. ID of device containing file
2. Inode number
3. File mode
4. Number of links

5 .UID of owner
6. GID of owner
7. ID of device containing directory entry for file
8. Size of file in bytes
9. Time of last access
10. Time of last modification
11. Time of last file status change
12. Preferred I/O block size (`-1` if not available)
13. Number of blocks allocated (`-1` if not available)

Except for elements 12 and 13, values are set to `0` if they are not available from the relevant file system.

ltime
Fortran interface to the C library function `localtime`. Sets `tarray` to the broken-down time corresponding to `stime`, which can be obtained from the intrinsic `time8`. All values are in the local time zone.

`tarray` must have nine elements:
1. Seconds since the last minute, ranging `0..61` (due to leap seconds)
2. Minutes since the last hour, ranging `0..59`
3. Hours since midnight, ranging `0..23`
4. Day of month, ranging `0..31`
5. Month, ranging `0..11`
6. Years since 1900
7. Days since Sunday, ranging `0..6`
8. Days since January 1, ranging `0..365`
9 . Positive if daylight savings time is in effect, zero if not, or negative if unknown

mclock, mclock8
Fortran interface to the C library function `clock`. Returns the number of clock ticks of CPU time since the start of execution of the process, or `-1` if this is not known.

or
Bitwise Boolean `OR`

perror
Like the C library function `perror`, prints on the stderr stream the `string` followed by a colon, a blank, and the message corresponding to the error code from the C library value `errno`.

rand
Fortran interface to POSIX function `rand`. Returns a uniform pseudorandom integer. If `flag` is `0`, return the next number in the current sequence; if `flag` is `1`, call POSIX function `srand(0)`; otherwise call `srand(flag)` to seed a new sequence.

realpart
Real part of a complex number (synonym for standard intrinsic `real`, which in Fortran 95 preserves the precision of its argument.)

rename
Fortran interface to the C library function `rename`. Change name of file `path1` to `path2`. The function form returns `0` on success or an error code from the C library value `errno`. The subroutine sets `status` to the value which the function would return.

Trailing blanks in `file` are ignored (you can prevent this by using `char(0)` to place a null character after the last significant character).

rshift
Arithmetic (sign-preserving) bitwise right shift. Shift count must be nonnegative and less than the bit-size of the data.

secnds
Return the number of seconds since midnight in the local time zone, minus the argument "`t`".

second
The function form returns the sum of user and system CPU time consumed by the process since the start of execution. The subroutine form sets `seconds` to that value.

short
Convert to type `integer*2`.

signal
Fortran interface to the C library function `signal`. Arrange for the signal whose number is `number` to trigger a call to external procedure `handler`, which should be a subroutine with no arguments; or restore the default response to the signal; or ignore the signal.

The optional third argument `igndfl` takes these values:
`-1` Use the second argument to provide a handler, to restore the default response to the signal, or to ignore the signal
`0` Regardless of the value of the second argument, restore the default response to the signal
`1` Regardless of the value of the second argument, ignore the signal

When `igndfl` is omitted, `handler` can instead be an integer, with these possible values:
(`address`) An integer containing the address of the external procedure
`0` Restore the default response to the signal
`1` Ignore the signal

The function form returns the previous state of the signal: zero (if the default response was in effect), one (if the signal was being ignored), or the address of a handler procedure.

Here is an example using the two-argument form:

```
C Keyboard interrupt (normally Control-C) alternately triggers
C handler1 and handler2 until 4 interrupts have occurred. Then
C restore the default handling, so the fifth interrupt stops the
C program.
      program once
        implicit none
        external handler1, handler2
        common previous, count
        integer*8 previous
        integer count
        previous = signal(2, handler1)
```

```
      previous = signal(2, handler2)
      count = 4
      do while (.true.)
        call sleep(100)
      end do
    end
    subroutine handler1()
      implicit none
      common previous, count
      integer*8 previous
      integer count
      print *, 'I am handler1'
      count = count - 1
      if (count .le. 0) then
        previous = 0
      end if
      previous = signal(2, previous)
    end subroutine handler1
    subroutine handler2()
      implicit none
      common previous, count
      integer*8 previous
      integer count
      print *, 'I am handler2'
      count = count - 1
      if (count .le. 0) then
        previous = 0
      end if
      previous = signal(2, previous)
    end subroutine handler2
```

Here is an example using the three-argument form:

```
C    Keyboard interrupt (normally Control-C) triggers handler
C    until 4 interrupts have occurred. Then restore the default,
C    so the fifth interrupt stops the program.
     program single
       implicit none
       external handler
       intrinsic signal
       integer*8 previous
       common count
       integer count
       previous = signal(2, handler, -1)
       count = 4
       do while (.true.)
         call sleep(100)
       end do
     end
     subroutine handler()
       implicit none
       intrinsic signal
       integer*8 previous
```

```
      common count
      integer count
      print *, 'I am handler'
      count = count - 1
      if (count .le. 0) then
        previous = signal(2, handler, 0)
      else previous = signal(2, handler, -1)
      end if
    end subroutine handler
```

sleep
Like the POSIX function `sleep`, pauses the process for `seconds` seconds.

srand
Like the POSIX function `srand`, restarts the random number sequence for `irand` or `rand` using `seed` as the seed.

stat
Fortran interface to the POSIX function `stat`. Store in array `sarray` information about the file named `file`; if that is a symbolic link, describe the target rather than the link itself (cf. `lstat`.). The function form returns `0`, or an error code from the C library value `errno`.

Trailing blanks in `file` are ignored (you can prevent this by using `char(0)` to place a null character after the last significant character).

`sarray` must have thirteen elements:
1. ID of device containing file
2. Inode number
3. File mode
4. Number of links
5. UID of owner
6. GID of owner
7. ID of device containing directory entry for file
8. Size of file in bytes
9. Time of last access
10. Time of last modification
11. Time of last file status change
12. Preferred I/O block size (-1 if not available)
13. Number of blocks allocated (-1 if not available)

Except for elements 12 and 13, values are set to `0` if they are not available from the relevant file system.

symlnk
Fortran interface to the POSIX function `symlink`. Creates a symbolic link `path2` pointing to the same file as `path1`. The function form returns `0` on success, or an error code from the C library value `errno`. The subroutine form sets `status` to the value which the function would return.

Trailing blanks in `path1` and `path2` are ignored (you can prevent this by using `char(0)` to place a null character after the last significant character).

system
Fortran interface to the C library function `system`. Execute `command` using a command interpreter or shell. The function form returns the value

returned by the interpreter (conventionally `0` to indicate success and nonzero to indicate failure). The subroutine form sets `status` to the value which the function would return.

time, time8
Fortran interface to the POSIX function `time`. Returns the current time as an integer suitable for use with `ctime`, `gmtime`, or `ltime`.

ttynam
Fortran interface to the POSIX function `ttyname`. The function form returns the name of the interactive terminal device associated with logical unit `unit`, or blanks if `unit` is not associated with such a device. The subroutine form sets `name` to the value that the function would return.

umask
Fortran interface to the POSIX function `umask`. Sets the file creation mask to `mask`. The function form returns the previous value of the mask. The subroutine form sets `old` to the previous value of the mask.

unlink
Fortran interface to the POSIX function `unlink`. Remove the link to the file named `file`. The function form returns `0` on success, or the error code from the C library value `errno`. The subroutine form sets `status` to the value which the function would return.

Trailing blanks in file are ignored (you can prevent this by using `char(0)` to place a null character after the last significant character.)

xor
Bitwise Boolean `XOR`

zabs, zcos, zexp, zlog, zsin, zsqrt
Specific names for various mathematical functions having an argument of type `complex*16.`

Appendix D

Fortran 90 dope vector

Here is an example of a simplified data structure from a Fortran 90 dope vector, from the file `clibinc/cray/dopevec.h` found in the source distribution. See Section 3.4.5 for more details.

```
typedef struct _FCD {
char *c_pointer; /* C character pointer */
unsigned long byte_len; /* Length of item (in bytes) */
} _fcd;
```

```
typedef struct f90_type {
    unsigned int :32; /* used for future development */
    enum typecodes {
        DVTYPE_UNUSED = 0,
        DVTYPE_TYPELESS = 1,
        DVTYPE_INTEGER = 2,
        DVTYPE_REAL = 3,
        DVTYPE_COMPLEX = 4,
        DVTYPE_LOGICAL = 5,
        DVTYPE_ASCII = 6,
        DVTYPE_DERIVEDBYTE = 7,
        DVTYPE_DERIVEDWORD = 8
    } type :8; /* type code */
    unsigned int dpflag :1; /* set if declared double precision
                                    * or double complex */
    enum dec_codes {
        DVD_DEFAULT = 0, /* KIND= and *n absent, or
                                     * KIND=expression which evaluates to
                                     * the default KIND, ie.:
                                     *      KIND(0) for integer
                                     *      KIND(0.0) for real
                                     *      KIND((0,0)) for complex
                                     *      KIND(.TRUE.) for logical
                                     *      KIND('A') for character
                                     * across on all ANSI-conformant
                                     *  implementations. */
        DVD_KIND = 1, /* KIND=expression which does not
                                      * qualify to be DVD_DEFAULT or
```

```
                                        * DVD_KIND_CONST or DVD_KIND_DOUBLE */
            DVD_STAR = 2, /* *n is specified (example: REAL*8 */
            DVD_KIND_CONST = 3, /* KIND=expression constant across
                                        * all implementations. */
            DVD_KIND_DOUBLE = 4 /* KIND=expression which evaluates to
                                        * KIND(1.0D0) for real across all
                                        * implementations.  This code may be
                                        * passed for real or complex type.  */
        } kind_or_star :3; /* Set if KIND= or *n appears in the
                                        * variable declaration.  Values
                                        * are from enum dec_codes */
        unsigned int int_len :12; /* internal length in bits of iolist
                                        * entity. 8 for character data to
                                        * indicate size of each character */
        unsigned int dec_len :8; /* declared length in bytes for *n
                                        * or KIND value. Ignored if
                                        * kind_or_star==DVD_DEFAULT */
    } f90_type_t;
    typedef struct DopeVector {
        union {
                    _fcd charptr; /* Fortran character descriptor */
                    struct {
                            void *ptr; /* pointer to base address */
                                                /* or shared data desc     */
                            unsigned long el_len; /* element length in bits  */
                    } a;
        } base_addr;
        /*
         * flags and information fields within word 3 of the header
         */
        unsigned int assoc :1; /* associated flag */
        unsigned int ptr_alloc :1; /* set if allocated by pointer */
        enum ptrarray {
            NOT_P_OR_A = 0,
            POINTTR = 1,
            ALLOC_ARRY = 2
        } p_or_a :2; /* pointer or allocatable array. Use */
                                         /* enum ptrarray values.  */
        unsigned int a_contig :1; /* array storage contiguous flag */
        unsigned int :27; /* pad for first 32 bits      */
        unsigned int :29; /* pad for second 32-bits      */
        unsigned int n_dim :3; /* number of dimensions */
        f90_type_t type_lens; /* data type and lengths */
        void *orig_base; /* original base address */
        unsigned long orig_size; /* original size */
        /*
         * Per Dimension Information - array will contain only the necessary
         * number of elements
         */
    #define MAXDIM 7
        struct DvDimen {
            signed long low_bound; /* lower bound for ith dimension */
                                         /* may be negative */
            signed long extent; /* number of elts for ith dimension */
            /*
```

```
            * The stride mult is not defined in constant units so that address
            * calculations do not always require a divide by 8 or 64.  For
            * double and complex, stride mult has a factor of 2 in it.  For
            * double complex, stride mult has a factor of 4 in it.
            */
           signed long stride_mult; /* stride multiplier */
       }dimension[7];
} DopeVectorType;
```

Appendix E

Reference: eko man page

There are online manual pages (“man pages”) available describing the flags and options for the PathScale EKOPath Compiler Suite. You can type "`man -k pathscale`" or "`apropos pathscale`" to get a list of all the PathScale man pages on your system. This feature does not work on SLES 8.

The following appendix is a copy of the information found in the `eko` man page, which is a listing of all of the supported flags and options.

You can view this same information online by typing:

```
$ man eko
```

The `eko` man page information begins on the following page.

NAME

eko - The complete list of options and flags for the PathScale(TM) EKOPath Compiler Suite

CG, **INLINE**, **IPA**, **LANG**, **LNO**, **OPT**, **TENV**, **WOPT** – other major topics covered

DESCRIPTION

This man page describes the various flags available for use with the PathScale EKOPath **pathcc**, **pathCC**, and **pathf95** compilers.

OPTIMIZATION FLAGS

Some suboptions either enable or disable the feature. To enable a feature, either specify only the suboption name or specify **=1**, **=ON**, or **=TRUE**. Disabling a feature, is accomplished by adding **=0**, **=OFF**, or **=FALSE**. These values are insensitive to case: 'on' and 'ON' mean the same thing. Below, **ON** and **OFF** are used to indicate the enabling or disabling of a feature.

Many options have an opposite ("no-") counterpart. This is represented as **[no-]** in the option description and if used, will turn off or prevent the action of the option. If no **[no-]** is shown, there is no opposite option to the listed option.

–### Like the **–v** option, only nothing is run and args are quoted.

–A pred=ans

Make an assertion with the predicate 'pred' and answer 'ans'. The **–pred=ans** form cancels an assertion with predicate 'pred' and answer 'ans'.

–align*N* Align data on common blocks to specified boundaries. The align*N* specifications are as follows:

Option	Action
-align8	Align data in common blocks to 8–bit boundaries.
-align16	Align data in common blocks to 16–bit boundaries.
-align32	Align data in common blocks 32–bit boundaries.
-align64	Align data in common blocks to 64–bit boundaries. This is the default.
-align128	Align data in common blocks to 128–bit boundaries.

When an alignment is specified, objects smaller than the specification are aligned on boundaries that correspond to their sizes. For example, when **align64** is specified, 32–bit and larger objects are aligned on 32–bit boundaries; 16–bit and larger objects are aligned on 16–bit boundaries; and 8–bit and larger objects are aligned on 8–bit boundaries.

–ansi (For **Fortran**) Cause the compiler to generate messages when it encounters source code that does not conform to the Fortran 90 standard. Specifying this option in conjunction with the **–fullwarn** option causes all messages, regardless of level, to be generated.

–ansi (For **C/C++**) Enable pure ANSI/ISO C mode.

–apo This auto-parallelizing option signals the compiler to automatically convert sequential code into parallel code by inserting parallel directives where it is safe and beneficial to do so.

–ar Create an archive using **ar**(1) instead of a shared object or executable. The name of the archive is specified by using the **–o** option. Template entities required by the objects being archived are instantiated before creating the archive. The **pathCC** command implicitly passes the **–r** and **–c** options of **ar** to **ar** in addition to the name of the archive and the objects being created. Any other option that can be used in conjunction with the **–c** option of **ar** can be passed to **ar** using **–WR,option_name**.

NOTE: The objects specified with this option must include all of the objects that will be included in the archive. Failure to do so may cause prelinker internal errors. In the following example, **liba.a** is an archive containing only **a.o**, **b.o**, and **c.o**. The **a.o**, **b.o**, and **c.o** objects are prelinked to instantiate any required template entities, and the **ar –r –c –v liba.a a.o b.o c.o** command is executed. All three objects must be specified with **–ar** even if only **b.o** needs to be replaced in **lib.a**.

```
pathCC -ar -WR,-v -o liba.a a.o b.o c.o
```

See the **ld**(1) man page for more information about shared libraries and archives.

−auto-use *module_name*[**,***module_name*] ...
(For **Fortran**) Direct the compiler to behave as if a **USE** *module_name* statement were entered in your Fortran source code for each *module_name*. The **USE** statements are entered in every program unit and interface body in the source file being compiled (for example, **pathf95 −auto-use mpi_interface** or **pathf95 −auto-use shmem_interface**). Using this option can add compiler time in some situations.

−backslash
Treat a backslash as a normal character rather than as an escape character. When this option is used, the preprocessor will not be called.

−C (For **Fortran**) Perform runtime subscript range checking. Subscripts that are out of range cause fatal runtime errors. If you set the **F90_BOUNDS_CHECK_ABORT** environment variable to **YES**, the program aborts.

−C (For **C**) Keep comments after preprocessing.

−c Create an intermediate object file for each named source file, but does not link the object files. The intermediate object file name corresponds to the name of the source file; a **.o** suffix is substituted for the suffix of the source file.

Because they are mutually exclusive, do not specify this option with the **−r** option.

−CG[:...]
The Code Generation option group controls the optimizations and transformations of the instruction−level code generator.

−CG:cflow=(ON|OFF)
OFF disables control flow optimization in the code generation. Default is **ON**.

−CG:cse_regs=N
When performing common subexpression elimination during code generation, assume there are **N** extra **integer** registers available over the number provided by the CPU. **N** can be positive, zero, or negative. The default is positive infinity. See also **-CG:sse_cse_regs**.

−CG:gcm=(ON|OFF)
Specifying **OFF** disables the instruction−level global code motion optimization phase. The default is **ON**.

−CG:load_exe=N
Specify the threshold for subsuming a memory load operation into the operand of an arithmetic instruction. The value of **0** turns off this subsumption optimization. If **N** is 1, this subsumption is performed only when the result of the load has only one use. This subsumption is not performed if the number of times the result of the load is used exceeds the value **N**, a non−negative integer. If the ABI is 64-bit and the language is Fortran, the default for **N** is 2, otherwise the default is 1. See also **-CG:sse_load_exe**.

−CG:local_fwd_sched=(ON|OFF)
Change the instruction scheduling algorithm to work forward instead of backward for the instructions in each basic block. The default is **OFF** for 64-bit ABI, and **ON** for 32-bit ABI.

−CG:movnti=N
Convert ordinary stores to non−temporal stores when writing memory blocks of size larger than **N** KB. When **N** is set to **0**, this transformation is avoided. The default value is **120 (KB)**.

−CG:p2align=(ON|OFF)
Align loop heads to 64-byte boundaries. The default is **OFF**.

−CG:p2align_freq=N
Align branch targets based on execution frequency. This option is meaningful only under feedback−directed compilation. The default value **N=0** turns off the alignment optimization. Any other value

specifies the frequency threshold at or above which this alignment will be performed by the compiler.

−CG:prefer_legacy_regs=(ON|OFF)
Tell the local register allocator to use the first 8 integer and SSE registers whenever possible (%rax-%rbp, %xmm0-%xmm7). Instructions using these registers have smaller instruction sizes. The default is **OFF**.

−CG:prefetch=(ON|OFF)
Suppress any generation of prefetch instructions in the code generator. This has the same effect as **-LNO:prefetch=0**. The default is **ON**.

−CG:sse_cse_regs=N
When performing common subexpression elimination during code generation, assume there are **N** extra **SSE** registers available over the number provided by the CPU. **N** can be positive, zero, or negative. The default is positive infinity. See also **-CG:cse_regs**.

−CG:sse_load_exe=N
This is similar to **-CG:load_exe** except that this only affects memory loads to the SSE co-processor. The default is 0. A memory load to the SSE is subsumed into an arithmetic instruction if it satisifies either the **-CG:sse_load_exe** or the **-CG:load_exe** condition.

−CG:use_prefetchnta=(ON|OFF)
Prefetch when data is non−temporal at all levels of the cache hierarchy. This is for data streaming situations in which the data will not need to be re-used soon. The default is **OFF**.

−CG:use_test=(ON|OFF)
Make the code generator use the TEST instruction instead of CMP. See Opteron's instruction description for the difference between these two instructions. The default is **OFF**.

−clist (**C** only) Enable the C listing. Specifying **−clist** is the equivalent of specifying **−CLIST:=ON**.

−CLIST: ...
(**C** only) Control emission of the compiler's internal program representation back into C code, after IPA inlining and loop−nest transformations. This is a diagnostic tool, and the generated C code may not always be compilable. The generated C code is written to two files, a header file containing file−scope declarations, and a file containing function definitions. With the exception of **−CLIST:=OFF**, any use of this option implies **−clist**. The individual controls in this group are as follows:

=(ON|OFF)
Enable the C listing. This option is implied by any of the others, but may be used to enable the listing when no other options are required. For example, specifying **−CLIST:=ON** is the equivalent of specifying **−clist**.

dotc_file=*filename*
Write the program units into the specified file, *filename*. The default source file name has the extension **.w2c.c**.

doth_file=*filename*
Specify the file into which file−scope declarations are deposited. Defaults to the source file name with the extension **.w2c.h**.

emit_pfetch[=(ON|OFF)]
Display prefetch information as comments in the transformed source. If **ON** or **OFF** is not specified, the default is **OFF**.

linelength=*N*
Set the maximum line length to *N* characters. The default is unlimited.

show[=(ON|OFF)]
Print the input and output file names to stderr. If **ON** or **OFF** is not specified, the default is **ON**.

−col*N* (**Fortran** only) Specify the line width for fixed−format source lines. Specify **72**, **80**, or **120** for *N* (-col72, -col80, or -col120). By default, fixed−format lines are 72 characters wide. Specifying **−col120** implies

−extend-source and recognizes lines up to 132 characters wide. For more information on specifying line length, see the **−extend-source** and **−noextend-source** options.

−copyright
Show the copyright for the compiler being used.

−cpp Run the preprocessor, **cpp**, on all input source files, regardless of suffix, before compiling. This preprocessor automatically expands macros outside of preprocessor statements.

The default is to run the C preprocessor (**cpp**) if the input file ends in a **.F** or **.F90** suffix.

For more information on controlling preprocessing, see the **−ftpp**, **−E**, and **−nocpp** options. For information on enabling macro expansion, see the **−macro-expand** option. By default, no preprocessing is performed on files that end in a **.f** or **.f90** suffix.

−d-lines (**Fortran** only) Compile lines with a D in column 1.

−D*var***=**[*def*][**,***var***=**[*def*] . . .]
Define variables used for source preprocessing as if they had been defined by a **#define** directive. If no *def* is specified, **1** is used. For information on undefining variables, see the **−U***var* option.

−default64
(For **Fortran** only) Set the sizes of default integer, real, logical, and double precision objects. This option causes the following options to go into effect: **−r8**, **−i8**, and **−64**. Calling a routine in a specialized library, such as SCSL, requires that its 64−bit entry point be specified when 64−bit data are used. Similarly, its 32−bit entry point must be specified when 32−bit data are used.

−dumpversion
Show the version of the compiler being used and nothing else.

−E Run only the source preprocessor files, without considering suffixes, and write the result to **stdout**. This option overrides the **−nocpp** option. The output file contains line directives. To generate an output file without line directives, see the **−P** option. For more information on controlling source preprocessing, see the **−cpp**, **−ftpp**, **−macro-expand**, and **−nocpp** options.

−extend-source
(For **Fortran** only) Specify a 132−character line length for fixed−format source lines. By default, fixed−format lines are 72 characters wide. For more information on controlling line length, see the **−col***n* option.

−fb-create <*path***>**
Used to specify that an instrumented executable program is to be generated. Such an executable is suitable for producing feedback data files with the specified prefix for use in feedback-directed compilation (FDO). The commonly used prefix is <*fbdata*>. This is **OFF** by default.

−fb-opt <prefix for feedback data files>
Used to specify feedback−directed compilation (FDO) by extracting feedback data from files with the specified prefix, which were previously generated using **−fb-create**. The commonly used prefix is "fbdata". The same optimization flags must have been used in the **−fb-create** compile. Feedback data files created from executables compiled with different optimization flags will give checksum errors. FDO is **OFF** by default.

-fb-phase=(0,1,2,3,4)
Used to specify the compilation phase at which instrumentation for the collection of profile data is performed, so is useful only when used with **−fb-create**. The values must be in the range 0 to 4. The default value is 0, and specifies the earliest phase for instrumentation, which is after the front-end processing.

−f[no-]check-new
(For **C++** only) Check the result of new for NULL. When **−fno−check−new** is used, the compiler will not check the result of an operator of NULL.

−fe Stop after the front-end is run.

–f[no-]unwind-tables

–funwind-tables emits unwind information. **–fno-unwind-tables** tells the compiler never to emit any unwind information. This is the default. Flags to enable exception handling automatically enable -funwind-tables.

–f[no-]fast–math

–ffast-math improves FP speed by relaxing ANSI & IEEE rules. **–ffast-math** is implied by **–Ofast**. **–fno-fast–math** tells the compiler to conform to ANSI and IEEE math rules at the expense of speed. **–ffast-math** implies **–OPT:IEEE_arithmetic=2 -fno-math-errno. -fno-fast-math** implies **-OPT:IEEE_arithmetic=1 -fmath-errno.**

–f[no-]fast–stdlib

The **–ffast-stdlib** flag improves application performance by generating code to link against special versions of some standard library routines, and linking against the EKOPath runtime library. This option is enabled by default.

If **–fno–fast–stdlib** is used during compilation, the compiler will not emit code to link against fast versions of standard library routines. During compilation, **–ffast–stdlib** implies **–OPT:fast_stdlib=on**.

If **–fno–fast–stdlib** is used during linking, the compiler will not link against the EKOPath runtime library.

If you link code with **–fno–fast–stdlib** that was not also compiled with this flag, you may see linker errors. Much of the EKOPath Fortran runtime is compiled with **–ffast–stdlib**, so it is not advised to link Fortran applications with **–fno-fast–stdlib**.

–ffloat–store

Do not store floating point variables in registers, and inhibit other options that might change whether a floating point value is taken from a register or memory. This option prevents undesirable excess precision on the X87 floating-point unit where all floating-point computations are performed in one precision regardless of the original type. (see **-mx87–precision**). If the program uses floating point values with less precision, the extra precision in the X87 may violate the precise definition of IEEE floating point. **-ffloat–store causes all pertinent immediate computations to be stored to memory to force truncation to lower precision. However, the extra stores will slow down program execution substantially.**

–ffortran-bounds-check

(For **Fortran** only) Check bounds.

–f[no-]gnu–keywords

(For **C/C++** only) Recognize 'typeof' as a keyword. If **–fno–gnu–keywords** is used, do not recognize 'typeof' as a keyword.

–f[no-]implicit-inline-templates

(For **C++** only) **–finplicit-inline-templates** emits code for inline templates instantiated implicitly. **–fno–implicit-inline-templates** tells the compiler to never emit code for inline templates instantiated implicitly.

–f[no-]implicit-templates

(For **C++** only) The **–finplicit-templates** option emits code for non–inline templates instantiated implicitly. With **–fno-implicit-templates** the compiler will not emit code for non–inline templates instantiated implicitly.

–finhibit-size-directive

Do not generate **.size** directives.

–f[no-]inline-functions

(For **C/C++** only) **–finline–functions** automatically integrates simple functions into their callers. **–fno-inline-functions** does not automatically integrate simple functions into their callers.

−fabi-version=N
(For **C++** only) Use version **N** of the C++ ABI. Version 1 is the version of the C++ ABI that first appeared in G++ 3.2. Version 0 will always be the version that conforms most closely to the C++ ABI specification. Therefore, the ABI obtained using version 0 will change as ABI bugs are fixed. The default is version **1**.

−fixedform
(For **Fortran** only) Treat all input source files, regardless of suffix, as if they were written in fixed source form (f77 72-column format), instead of F90 free format. By default, only input files suffixed with **.f** or

−fkeep-inline-functions
(For **C/C++** only) Generate code for functions even if they are fully inlined.

−FLIST: . . .
Invoke the Fortran listing control group, which controls production of the compiler's internal program representation back into Fortran code, after IPA inlining and loop−nest transformations. This is used primarily as a diagnostic tool, and the generated Fortran code may not always compile. With the exception of **−FLIST:=OFF**, any use of this option implies **−flist**. The arguments to the **−FLIST** option are as follows:

Argument
Action

=*setting* Enable or disable the listing. **setting** can be either **ON** or **OFF**. The default is **OFF**.

This option is enabled when any other **−FLIST** options are enabled, but it can also be used to enable a listing when no other options are enabled.

ansi_format=*setting*
Set ANSI format. **setting** can be either **ON** or **OFF**. When set to **ON**, the compiler uses a space (instead of tab) for indentation and a maximum of 72 characters per line. The default is **OFF**.

emit_pfetch=setting
Writes prefetch information, as comments, in the transformed source file. *setting* can be either **ON** or **OFF**. The default is **OFF**.

In the listing, **PREFETCH** identifies a prefetch and includes the variable reference (with an offset in bytes), an indication of read/write, a stride for each dimension, and a number in the range from 1 (low) to 3 (high), which reflects the confidence in the prefetch analysis. Prefetch identifies the reference(s) being prefetched by the **PREFETCH** descriptor. The comments occur after a read/write to a variable and note the identifier of the PREFETCH−spec for each level of the cache.

ftn_file=*file*
Write the program to *file*. By default, the program is written to *file***.w2f.f**.

linelength=*N*
Set the maximum line length to *N* characters.

show=*setting*
Write the input and output filenames to **stderr**. *setting* can be either **ON** or **OFF**. The default is **ON**.

−flist Invoke all Fortran listing control options. The effect is the same as if all **−FLIST** options are enabled.

−fms-extensions
(For **C/C++** only) Accept broken MFC extensions without warning.

−fno-asm
(For **C/C++** only) Do not recognize the 'asm' keyword.

−fno-builtin
(For **C/C++** only) Do not recognize any built in functions.

−fno-common
(For **C/C++** only) Use strict ref/def initialization model.

−f[no-]exceptions
(For **C++** only) **−fexceptions** enables exception handling. This is the default. **−fno-exceptions** disables exception handling.

−f[no-]fast−math
−ffast-math improves FP speed by relaxing ANSI & IEEE rules. **−fno-fast−math** tells the compiler to conform to ANSI and IEEE math rules at the expense of speed.

−f[no-]gnu−keywords
(For **C/C++** only) Recognize 'typeof' as a keyword. If **−fno−gnu−keywords** is used, do not recognize 'typeof' as a keyword.

−fno-ident
Ignore #ident directives.

−fno−math−errno
Do not set ERRNO after calling math functions that are executed with a single instruction, e.g. **sqrt**. A program that relies on IEEE exceptions for math error handling may want to use this flag for speed while maintaining IEEE arithmetic compatibility. This is implied by **−Ofast**. The default is **−fmath-errno**.

−f[no−]signed−char
(For **C/C++** only) **−fsigned−char** makes 'char' signed by default. **−fno−signed−char** makes 'char' unsigned by default.

−fpack-struct
(For **C/C++** only) Pack structure members together without holes.

−f[no−]permissive
−fpermissive will downgrade messages about non−conformant code to warnings. **−fno−permissive** keeps messages about non−conformant code as errors.

−f[no−]preprocessed
−fpreprocessed tells the preprocessor that input has already been preprocessed. Using **−fno−preprocessed** tells preprocessor that input has not already been preprocessed.

−freeform
(For **Fortran** only) Treats all input source files, regardless of suffix, as if they were written in free source form. By default, only input files suffixed with **.f90** or **.F90** are assumed to be written in free source form.

−f[no-]rtti
(For **C++** only) Using **−frtti** will generate runtime type information. The **−fno-rtti** option will not generate runtime type information.

−f[no-]second-underscore
(For **Fortran** only) **−fsecond-underscore** appends a second underscore to symbols that already contain an underscore. **−fno−second-underscore** tells the compiler not to append a second underscore to symbols that already contain an underscore.

−f[no-]signed-bitfields
(For **C/C++** only) **−fsigned-bitfields** makes bitfields be signed by default. The **−fno-signed-bitfields** will make bitfields be unsigned by default.

−f[no-]strict-aliasing
(For **C/C++** only) **−fstrict−aliasing** tells the compiler to assume strictest aliasing rules. **−fno−strict−aliasing** tells the compiler not to assume strict aliasing rules.

−f[no-]PIC
−fPIC tells the compiler to generate position independent code, if possible. The default is **−fno−PIC**, which tells the compiler not to generate position independent code.

−fprefix-function-name
(For **C/C++** only) Add a prefix to all function names.

−fshared-data
(For **C/C++** only) Mark data as shared rather than private.

−fshort-double
(For **C/C++** only) Use the same size for double as for float.

−fshort-enums
(For **C/C++** only) Use the smallest fitting integer to hold enums.

−fshort-wchar
(For **C/C++** only) Use short unsigned int for wchar_t instead of the default underlying type for the target.

−ftest-coverage
Create data files for the **pathcov**(1) code-coverage utility. The data file names begin with the name of your source file:

SOURCENAME.bb
A mapping from basic blocks to line numbers, which **pathcov** uses to associate basic block execution counts with line numbers.

SOURCENAME.bbg
A list of all arcs in the program flow graph. This allows **pathcov** to reconstruct the program flow graph, so that it can compute all basic block and arc execution counts from the information in the **SOURCENAME.da** file.

Use **−ftest-coverage** with **−fprofile-arcs**; the latter option adds instrumentation to the program, which then writes execution counts to another data file:

SOURCENAME.da
Runtime arc execution counts, used in conjunction with the arc information in the file **SOURCENAME.bbg**.

Coverage data will map better to the source files if **−ftest-coverage** is used without optimization. See the gcc man pages for more information.

−ftpp Run the Fortran source preprocessor on input Fortran source files before compiling. By default, files suffixed with **.F** or **.F90** are run through the C source preprocessor (**cpp**). Files that are suffixed with **.f** or **.f90** are not run through any preprocessor by default.

The Fortran source preprocessor does not automatically expand macros outside of preprocessor statements, so you need to specify **−macro-expand** if you want macros expanded.

−fullwarn
Request that the compiler generate comment−level messages. These messages are suppressed by default. Specifying this option can be useful during software development.

−f[no-]underscoring
(For **Fortran** only) **−funderscoring** appends underscores to symbols. **−fno-underscoring** tells the compiler not to append underscores to symbols.

−f[no-]unsafe-math-optimizations
−funsafe-math-optimizations improves FP speed by violating ANSI and IEEE rules. **−fno-unsafe-math-optimizations** makes the compilation conform to ANSI and IEEE math rules at the expense of speed. This option is provided for GCC compatibility and is equivalent to **−OPT:IEEE_arithmetic=3 −fno−math−errno**.

−fuse-cxa-atexit
(For **C++** only) Register static destructors with **__cxa_atexit** instead of **atexit**.

−fwritable-strings
(For **C/C++** only) Attempt to support writable-strings K&R style C.

−g[*N*] Specify debugging support and to indicate the level of information produced by the compiler. The supported values for **N** are:

0 No debugging information for symbolic debugging is produced. This is the default.

1 Produces minimal information, enough for making backtraces in parts of the program that you don't plan to debug. This is also the flag to use if the user wants backtraces but does not want the overhead of full debug information. This flag also causes **−−export−dynamic** to be passed to the linker.

2 Produces debugging information for symbolic debugging. Specifying **−g** without a debug level is equivalent to specifying **−g2**. If there is no explicit optimization flag specified, the **−O0** optimization level is used in order to maintain the accuracy of the debugging information. If optimization options **−O1**, **−O2**, **−O3** or **−ipa** are explicitly specified, the optimizations are performed accordingly but the accuracy of the debugging cannot be guaranteed.

3 Produces additional debugging information for debugging macros.

−gcc (For **C/C++** only) Define the **__GNUC__** and other predefined preprocessor macros.

−GRA:home=(ON|OFF)
Turn off the rematerialization optimization for non−local user variables in the Global Register Allocator. Default is **ON**.

−GRA:optimize_boundary=(ON|OFF)
Allow the Global Register Allocator to allocate the same register to different variables in the same basic-block. Default is **OFF**.

−help List all available options. The compiler is not invoked.

−help: Print list of possible options that contain a given string.

−H Print the name of each header file used.

−I*dir* Specify a directory to be searched. This is used for the following types of files:

- Files named in **INCLUDE** lines in the Fortran source file that do not begin with a slash (**/**) character
- Files named in **#include** source preprocessing directives that do not begin with a slash (**/**) character
- Files specified on Fortran **USE** statements

Files are searched in the following order: first, in the directory that contains the input file; second, in the directories specified by *dir*; and third, in the standard directory, **/usr/include**.

−i*N* (For **Fortran** only) Specify the length of default integer constants, default integer variables, and logical quantities. Specify one of the following:

Option	Action
−i4	Specifies 32−bit (4 byte−) objects. The default.
−i8	Specifies 64−bit (8 byte−) objects.

−ignore-suffix
Determine the language of the source file being compiled by the command used to invoke the compiler. By default, the language is determined by the file suffixes (**.c**, **.cpp**, **.C**, **.cxx**, **.f**, **.f90**, **.s**). When the **−ignore-suffix** option is specified, the **pathcc** command invokes the C compiler, **pathCC** invokes the C++ compiler, and **pathf95** invokes the Fortran 95 compiler.

−inline Request inline processing.

−INLINE: ...
Specify options for subprogram inlining. may not always compile. With the exception of **−INLINE:=OFF**, any use of this option implies **−inine**.

If you have included inlining directives in your source code, the **−INLINE** option must be specified in order for those directives to be honored.

−INLINE:aggressive=(ON|OFF)
Tell the compiler to be more aggressive about inlining. The default is **−INLINE:aggressive=OFF**.

−INLINE:list=(ON|OFF)
Tell the inliner to list inlining actions as they occur to **stderr**. The default is **−INLINE:list=OFF**.

−INLINE:preempt=(ON|OFF)
Perform inlining of functions marked preemptible in the light-weight inliner. Default is **OFF**. This inlining prevents another definition of such a function, in another DSO, from preempting the definition of the function being inlined.

−ipa Invoke inter-procedural analysis (IPA). Specifying this option is identical to specifying **−IPA** or **−IPA:**. Default settings for the individual IPA suboptions are used.

−IPA: ...
The inter-procedural analyzer option group controls application of inter-procedural analysis and optimization, including inlining, constant propagation, common block array padding, dead function elimination, alias analysis, and others. Specify **−IPA** by itself to invoke the inter-procedural analysis phase with default options. If you compile and link in distinct steps, you must specify at least **−IPA** for the compile step, and specify **−IPA** and the individual options in the group for the link step. If you specify **−IPA** for the compile step, and do not specify **−IPA** for the link step, you will receive an error.

−IPA:addressing=(ON|OFF)
Invoke the analysis of address operator usage. The default is **Off**. **−IPA:alias=ON** is a prerequisite for this option.

−IPA:aggr_cprop=(ON|OFF)
Enable or disable aggressive inter-procedural constant propagation. Setting can be **ON** or **OFF**. This attempts to avoid passing constant parameters, replacing the corresponding formal parameters by the constant values. Less aggressive inter-procedural constant propagation is done by default. The default setting is **ON**.

−IPA:alias=(ON|OFF)
Invoke alias/mod/ref analysis. The default is **ON**.

−IPA:callee_limit=N
Functions whose size exceeds this limit will never be automatically inlined by the compiler. The default is **500**.

−IPA:cgi=(ON|OFF)
Invoke constant global variable identification. This option marks non-scalar global variables that are never modified as constant, and propagates their constant values to all files. Default is **ON**.

−IPA:common_pad_size=N
This specifies the amount by which to pad common block array dimensions. By default, an amount is automatically chosen that will improve cache behavior for common block array accesses.

−IPA:cprop=(ON|OFF)
Turn on or off inter-procedural constant propagation. This option identifies the formal parameters that always have a specific constant value. Default is **ON**. See also **-IPA:aggr_cprop**.

−IPA:ctype=(ON|OFF)
When **ON**, causes the compiler to generate faster versions of the <ctype.h> macros such as isalpha, isascii, etc. This flag is unsafe both in multi-threaded programs and in all locales other than the 7-bit ASCII (or "C") locale. The default is **OFF**. Do not turn this on unless the program will always run under the 7-bit ASCII (or "C") locale and is single-threaded.

−IPA:depth=N
Identical to **maxdepth=N**.

−IPA:dfe=(ON|OFF)
Enable or disable dead function elimination. Removes any functions that are inlined everywhere they are called. The default is **ON**.

−IPA:dve=(ON|OFF)
Enable or disable dead variable elimination. This option removes variables that are never referenced by the program. Default is **ON**.

−IPA:echo=(ON|OFF)
Option to echo (to stderr) the compile commands and the final link commands that are invoked from IPA. Default is **OFF**. This option can help monitor the progress of a large system build.

−IPA:field_reorder=(ON|OFF)
Enable the re−ordering of fields in large structs based on their reference patterns in feedback compilation to minimize data cache misses. The default is **OFF**.

−IPA:forcedepth=N
This option sets inline depths, directing IPA to attempt to inline all functions at a depth of (at most) **N** in the callgraph, instead of using the default inlining heuristics. This option ignores the default heuristic limits on inlining. Functions at depth 0 make no calls to any sub-functions. Functions only making calls to depth 0 functions are at depth 1, and so on.

−IPA:inline=(ON|OFF)
This option performs inter-file subprogram inlining during the main IPA processing. The default is **ON**. Does not affect the light-weight inliner.

−IPA:keeplight=(ON|OFF)
This option directs IPA not to send **−keep** to the compiler, in order to save space. The default is **OFF**.

−IPA:linear=(ON|OFF)
Controls conversion of a multi-dimensional array to a single dimensional (linear) array that covers the same block of memory. When inlining Fortran subroutines, IPA tries to map formal array parameters to the shape of the actual parameter. In the case that it cannot map the parameter, it linearizes the array reference. By default, IPA will not inline such callsites because they may cause performance problems. The default is **OFF**.

−IPA:map_limit=N
Direct when IPA enables **sp_partition**. **N** is the maximum size (in bytes) of input files mapped before IPA invokes **-IPA:sp_partition**.

−IPA:maxdepth=N
This option directs IPA to not attempt to inline functions at a depth of more than **N** in the callgraph; where functions that make no calls are at depth 0, those that call only depth 0 functions are at depth 1, and so on. This inlining remains subject to overriding limits on code expansion. Also see **−IPA:forcedepth**, **−IPA:space**, and **−IPA:plimit**.

−IPA:max_jobs=N
This option limits the maximum parallelism when invoking the compiler after IPA to (at most) **N** compilations running at once. The option can take the following values:

0 = The parallelism chosen is equal to either the number of CPUs,
the number of cores, or the number of hyperthreading units in the compiling system, whichever is greatest.

1 = Disable parallelization during compilation (default)

>1 = Specifically set the degree of parallelism

−IPA:min_hotness=N
When feedback information is available, a call site to a procedure must be invoked with a count that exceeds the threshold specified by **N** before the procedure will be inlined at that call site. The default is 10.

−IPA:multi_clone=N
This option specifies the maximum number of clones that can be created from a single procedure. Default value is **0**. Aggressive procedural cloning may provide opportunities for inter-procedural optimization, but may also significantly increase the code size.

−IPA:clone_list=(ON|OFF)
Tell the IPA function cloner to list cloning actions as they occur to **stderr**. The default is **−IPA:clone_list=OFF**.

−IPA:node_bloat=N
When this option is used in conjunction with **−IPA:multi_clone**, it specifies the maximum percentage growth of the total number of procedures relative to the original program.

−IPA:plimit=N
This option stops inlining into a specific subprogram once it reaches size **N** in the intermediate representation. Default is **2500**.

−IPA:pu_reorder=(0|1|2)
Control re−ordering the layout of program units based on their invocation patterns in feedback compilation to minimize instruction cache misses. This option is ignored unless under feedback compilation.

0 = Disable procedure reordering. This is the default for non−C++ programs.

1 = Reorder based on the frequency in which different procedures are invoked. This is the default for C++ programs.

2 = Reorder based on caller-callee relationship.

−IPA:relopt=(ON|OFF)
This option enables optimizations similar to those achieved with the compiler options **−O** and **−c**, where objects are built with the assumption that the compiled objects will be linked into a call-shared executable later. The default is **OFF**. In effect, optimizations based on position-dependent code (non-PIC) are performed on the compiled objects.

−IPA:small_pu=N
A procedure with size smaller than **N** is not subjected to the plimit restriction. The default is 30.

−IPA:sp_partition=[setting]
This option enables partitioning for disk/addressing−saving purposes. The default is **OFF**. Mainly used for building very large programs. Normally, partitioning would be done by IPA internally.

−IPA:space=N
Inline until a program expansion of **N%** is reached. For example, **-IPA:space=20** limits code expansion due to inlining to approximately 20%. Default is no limit.

−IPA:specfile=*file name*
Opens a *file name* to read additional options. The specification file contains zero or more lines with inliner options in the form expected on the command line. The specfile option cannot occur in a specification file, so specification files cannot invoke other specification files.

−IPA:use_intrinsic=(ON|OFF)
Enable/disable loading the intrinsic version of standard library functions. The default is **OFF**.

−isystem dir
Search dir for header files, after all directories specified by **−I** but before the standard system directories. Mark it as a system directory, so that it gets the same special treatment as is applied to the standard system directories.

−keep Write all intermediate compilation files. *file***.s** contains the generated assembly language code. *file***.i** contains the preprocessed source code. These files are retained after compilation is finished. If IPA is in effect and you want to retain *file***.s**, you must specify **−IPA:keeplight=OFF** in addition to **−keep**.

−keepdollar
(For **Fortran** only) Treat the dollar sign ($) as a normal last character in symbol names.

−L *directory*
In XPG4 mode, changes the algorithm of searching for libraries named in **−L** operands to look in the specified directory before looking in the default location. Directories specified in **−L** options are searched in the specified order. Multiple instances of **−L** options can be specified.

−l *library*
In XPG4 mode, searches the specified *library*. A library is searched when its name is encountered, so the placement of a **−l** operand is significant.

−LANG: . . .
Controls the language option group. The following sections describe the suboptions available in this group.

Argument
Action

copyinout=(ON|OFF)
When an array section is passed as the actual argument in a call, the compiler sometimes copies the array section to a temporary array and passes the temporary array, thus promoting locality in the accesses to the array argument. This optimization is relevant only to Fortran, and this flag controls the aggressiveness of this optimization. The default is **ON** for **−O2** or higher and **OFF** otherwise.

formal_deref_unsafe=(ON|OFF)
Tell the compiler whether it is unsafe to speculate a dereference of a formal parameter in Fortran. The default is **OFF**, which is better for performance.

heap_allocation_threshold=*size*
Determine heap or stack allocation. If the size of an automatic array or compiler temporary exceeds *size* bytes it is allocated on the heap instead of the stack. If *size* is **−1**, objects are always put on the stack. If *size* is **0**, objects are always put on the heap.

The default is **−1** for maximum performance and for compatibility with previous releases.

recursive=*setting*
Invoke the language option control group to control recursion support. *setting* can be either **ON** or **OFF**. The default is **OFF**.

In either mode, the compiler supports a recursive, stack−based calling sequence. The difference lies in the optimization of statically allocated local variables, as described in the following paragraphs.

With **−LANG:recursive=ON**, the compiler assumes that a statically allocated local variable could be referenced or modified by a recursive procedure call. Therefore, such a variable must be stored into memory before making a call and reloaded afterwards.

With **−LANG:recursive=OFF**, the compiler can safely assume that a statically allocated local variable is not referenced or modified by a procedure call. This setting enables the compiler to optimize more aggressively.

rw_const=(ON|OFF)
Tell the compiler whether to treat a constant parameter in Fortran as read-only or read-write. If treated as read-write, the compiler has to generate extra code in passing these constant parameters so as to tolerate their being modified in the called function. The default is **OFF**, which is more efficient but will cause segmentation fault if the constant parameter is written into.

short_circuit_conditionals=(ON|OFF)
Handle **.AND.** and **.OR.** via short-circuiting, in which the second operand is not evaluated if unnecessary, even if it contains side effects. Default is **ON**. This flag is applicable only to Fortran, the flag has no effect on C/C++ programs.

−LIST: ...
The listing option flag controls information that gets written to a listing (**.lst**) file. The individual controls in this group are:

=(ON|OFF)
Enable or disable writing the listing file. The default is **ON** if any **−LIST:** group options are enabled. By default, the listing file contains a list of options enabled.

all_options[=(ON|OFF)]
Enable or disable listing of most supported options. The default is **OFF**.

notes[=(ON|OFF)]
If an assembly listing is generated (for example, on **−S**), various parts of the compiler (such as software pipelining) generate comments within the listing that describe what they have done. Specifying **OFF** suppresses these comments. The default is **ON**.

options[=(ON|OFF)]
Enable or disable listing of the options modified (directly in the command line, or indirectly as a side effect of other options). The default is **OFF**.

symbols[=(ON|OFF)]
Enable or disable listing of information about the symbols (variables) managed by the compiler.

−LNO: ...
Specify options and transformations performed on loop nests by the Loop Nest Optimizer (LNO). The **−LNO** options are enabled only if **−O3** is also specified on the **pathf95**(1) command line.

For information on the LNO options that are in effect during a compilation, use the **−LIST:all_options=ON** option.

−LNO:apo_use_feedback=(ON|OFF)
Effective only when specified with **−apo** under feedback−directed compilation, this flag tells the auto-parallelizer whether to use the feedback data of the loops in deciding whether each loop should be parallelized. When the compiler parallelizes a loop, it generates both a serial and a parallel version. If the trip count of the loop is small, it is not beneficial to use the parallel version during execution. When this flag is set to **ON** and the feedback data indicates that the loop has small trip count, the auto−parallelizer will not generate the parallel version, thus saving the runtime check needed to decide whether to execute the serial or parallel version of the loop. The default is **OFF**.

−LNO:build_scalar_reductions=(ON|OFF)
Build scalar reductions before any loop transformation analysis. Using this flag may enable further loop transformations involving reduction loops. The default is **OFF**. This flag is redundant when -OPT:roundoff=2 or greater is in effect.

−LNO:blocking=(ON|OFF)
Enable or disable the cache blocking transformation. The default is **ON** at **−O3** or higher.

−LNO:blocking_size=N
This option specifies a block size that the compiler must use when performing any blocking. **N** must be a positive integer number that represents the number of iterations.

−LNO:fission=(0|1|2)
This option controls loop fission. The options can be one of the following:

0 = Disable loop fission (default)

1 = Perform normal fission as necessary

2 = Specify that fission be tried before fusion

Because -LNO:fusion is on by default, turning on fission without turning off fusion may result in their effects being nullified. Ordinarily, fusion is applied before fission. Specifying **-LNO:fission=2** will turn on fission and cause it to be applied before fusion.

–LNO:full_unroll,fu=N
Fully unroll loops with trip_count <= **N** inside LNO. **N** can be any integer between 0 and 100. The default value for **N** is 5. Setting this flag to 0 disables full unrolling of small trip count loops inside LNO.

–LNO:full_unroll_size=N
Fully unroll loops with unrolled loop size <= **N** inside LNO. **N** can be any integer between 0 and 10000. The conditions implied by the **full_unroll** option must also be satisfied for the loop to be fully unrolled. The default value for **N** is 2000.

–LNO:full_unroll_outer=(ON|OFF)
Control the full unrolling of loops with known trip count that do not contain a loop and are not contained in a loop. The conditions implied by both the **full_unroll** and the **full_unroll_size** options must be satisfied for the loop to be fully unrolled. The default is **OFF**.

–LNO:fusion=N
Perform loop fusion. **N** can be one of the following:

0 = Loop fusion is off

1 = Perform conservative loop fusion

2 = Perform aggressive loop fusion

The default is **1**.

–LNO:fusion_peeling_limit=N
This option sets the limit for the number of iterations allowed to be peeled in fusion, where **N**>= 0. **N**=5 by default.

–LNO:gather_scatter=N
This option enables gather-scatter optimizations. **N** can be one of the following:

0 = Disable all gather-scatter optimizations

1 = Perform gather-scatter optimizations in non-nested IF statements (default)

2 = Perform multi-level gather-scatter optimizations

–LNO:hoistif=(ON|OFF)
This option enables or disables hoisting of IF statements inside inner loops to eliminate redundant loops. Default is **ON**.

–LNO:ignore_feedback=(ON|OFF)
If the flag is **ON** then feedback information from the loop annotations will be ignored in LNO transformations. The default is **OFF**.

–LNO:ignore_pragmas=(ON|OFF)
This option specifies that the command-line options override directives in the source file. Default is **OFF**.

–LNO:local_pad_size=N
This option specifies the amount by which to pad local array dimensions. The compiler automatically (by default) chooses the amount of padding to improve cache behavior for local array accesses.

–LNO:non_blocking_loads=(ON|OFF)
(For **C/C++** only) The option specifies whether the processor blocks on loads. If not set, the default of the current processor is used.

–LNO:oinvar=(ON|OFF)
This option controls outer loop hoisting. Default is **ON**.

–LNO:opt=(0|1)
This option controls the LNO optimization level. The options can be one of the following:

0 = Disable nearly all loop nest optimizations.

1 = Perform full loop nest transformations. This is the default.

–LNO:ou_prod_max=N
This option indicates that the product of unrolling of the various outer loops in a given loop nest is not to exceed **N**, where **N** is a positive integer. The default is **16**.

–LNO:outer=(ON|OFF)
This option enables or disables outer loop fusion. Default is **ON**.

–LNO:outer_unroll_max,ou_max=N
The **Outer_unroll_max** option indicates that the compiler may unroll outer loops in a loop nest by as many as **N** per loop, but no more. The default is **4**.

–LNO:parallel_overhead=N
Effective only when specified with **-apo**, the **parallel_overhead** option controls the auto-parallelizing compiler's estimate of the overhead (in processor cycles) incurred by invoking the parallel version of a loop. When the compiler parallelizes a loop, it generates both a serial and a parallel version. If the amount of work performed by the loop is small, it may not be beneficial to use the parallel version during execution. The set value of **parallel_overhead** is used in this determination during execution time when the number of processors and the iteration count of the loop are taken into account. The default value is **4096**. Because the optimal value varies across systems and programs, this option can be used for parallel performance tuning.

–LNO:prefetch=(0|1|2|3)
This option specifies the level of prefetching.

0 = Prefetch disabled.

1 = Prefetch is done only for arrays that are always referenced in each iteration of a loop.

2 = Prefetch is done without the above restriction. This is the default.

3 = Most aggressive.

–LNO:prefetch_ahead=N
Prefetch **N** cache line(s) ahead. The default is **2**.

–LNO:prefetch_verbose=(ON|OFF)
–LNO:prefetch_verbose=ON prints verbose prefetch info to stdout. Default is **OFF**.

–LNO:processors=N
Tells the compiler to assume that the program compiled under **-apo** will be run on a system with the given number of processors. This helps in reducing the amount of computation during execution for determining whether to enter the parallel or serial versions of loops that are parallelized (see the **–LNO:parallel_overhead** option). The default is **0**, which means unknown number of processors. The default value of **0** should be used if the program is intended to run in different systems with different number of processors. If the option is set to non-zero and the value is different from the number of processors, the parallelized code will not perform optimally.

–LNO:sclrze=(ON|OFF)
Turn **ON** or **OFF** the optimization that replaces an array by a scalar variable. The default is **ON**.

–LNO:simd=(0|1|2)
This option enables or disables inner loop vectorization.

0 = Turn off the vectorizer.

1 = (Default) Vectorize only if the compiler can determine that there is no undesirable performance impact due to sub-optimal alignment. Vectorize only if vectorization does not introduce accuracy problems with floating-point operations.

2 = Vectorize without any constraints (most aggressive).

–LNO:simd_reduction=(ON|OFF)
This controls whether reduction loops will be vectorized. Default is **ON**.

–LNO:simd_verbose=(ON|OFF)
–LNO:simd_verbose=ON prints verbose vectorizer info to stdout. Default is **OFF**.

–LNO:svr_phase1=(ON|OFF)
This flag controls whether the scalar variable naming phase should be invoked before first phase of LNO. The default is **ON**.

–LNO:vintr=(0|1|2)
–LNO:vintr=1 is the default. **–LNO:vintr=0** will turn off vectorization of math intrinsics. Under **–LNO:vintr=2** the compiler will vectorize all math functions. Note that **vintr=2** could be unsafe in that the vector forms of some of the functions could have accuracy problems.

–LNO:vintr_verbose=(ON|OFF)
–LNO:vinter_verbose=ON prints verbose info to stdout on vectorizing math functions. Default is **OFF**. This flag will let you know that the intrinsics are not vectorized.

Following are **LNO Transformation Options.** Loop transformation arguments allow control of cache blocking, loop unrolling, and loop interchange. They include the following options.

–LNO:interchange=(ON|OFF)
Disable the loop interchange transformation in the loop nest optimizer. Default is **ON**.

–LNO:unswitch=(ON|OFF)
Turn **ON** or **OFF** the optimization that performs a simple form of loop unswitching. The default is **ON**.

–LNO:unswitch_verbose=(ON|OFF)
–LNO:unswitch_verbose=ON prints verbose info to stdout on unswitching loops. Default is **OFF**.

–LNO:ou=N
This option indicates that all outer loops for which unrolling is legal should be unrolled by **N**, where **N** is a positive integer. The compiler unrolls loops by this amount or not at all.

–LNO:ou_deep=(ON|OFF)
This option specifies that for loops with 3-deep (or deeper) loop nests, the compiler should outer unroll the wind-down loops that result from outer unrolling loops further out. This results in large code size, but generates faster code (whenever wind-down loop execution costs are important). Default is **ON**.

–LNO:ou_further=N
This option specifies whether or not the compiler performs outer loop unrolling on wind-down loops. **N** must be specified and be an integer.

Additional unrolling can be disabled by specifying **–LNO:ou_further=999999**. Unrolling is enabled as much as is sensible by specifying **–LNO:ou_further=3**.

–LNO:ou_max=N
This option enables the compiler to unroll as many as **N** copies per loop, but no more.

–LNO:pwr2=(ON|OFF)
(For **C/C++** only) This option specifies whether to ignore the leading dimension (set this to **OFF** to ignore).

Following are **LNO Target Cache Memory Options.** These arguments allow you to describe the target cache memory system. In the following arguments, the numbering starts with the cache level closest to the processor and works outward.

–LNO:assoc1=N, assoc2=N, assoc3=N, assoc4=N
This option specifies the cache set associativity. For a fully associative cache, such as main memory, **N** should be set to any sufficiently large number, such as 128. Specify a positive integer for **N**; specifying **N=0** indicates there is no cache at that level.

–LNO:cmp1=N, cmp2=N, cmp3=N, cmp4=N, dmp1=N, dmp2=N, dmp3=N, dmp4=N
This option specifies, in processor cycles, the time for a clean miss (**cmp**x=) or a dirty miss (**dmp**x=) to the next outer level of the memory hierarchy. This number is approximate because it depends on a clean or dirty line, read or write miss, etc. Specify a positive integer for **N**; specifying **N=0** indicates there is no cache at that level.

–LNO:cs1=N, cs2=N, cs3=N, cs4=N
This option specifies the cache size. **N** can be 0 or a positive integer followed by one of the following letters: k, K, m, or M. These letters specify the cache size in Kbytes or Mbytes. Specifying **0** indicates there is no cache at that level.

cs1 is the primary cache, **cs2** refers to the secondary cache, **cs3** refers to memory, and **cs4** is the disk. Default cache size for each type of cache depends on your system. Use **–LIST:options=ON** to see the default cache sizes used during compilation.

–LNO:is_mem1=(ON|OFF), is_mem2=(ON|OFF), is_mem3=(ON|OFF), is_mem4=(ON|OFF)
This option specifies that certain memory hierarchies should be modeled as memory not cache. Default is **OFF** for each option.

Blocking can be attempted for this memory level, and blocking appropriate for memory, rather than cache, is applied. No prefetching is performed, and any prefetching options are ignored. If **–OPT:is_memx=(ON|OFF)** is specified, the corresponding **assocx=N** specification is ignored, any **cmpx=N** and **dmpx=N** options on the command line are ignored.

–LNO:ls1=N, ls2=N, ls3=N, ls4=N
This option specifies the line size in bytes. This is the number of bytes, specified in the form of a positive integer number (**N**), that are moved from the memory hierarchy level further out to this level on a miss. Specifying **N=0** indicates there is no cache at that level.

Following are **LNO TLB Options.** These arguments control the TLB, a cache for the page table, assumed to be fully associative. The TLB control arguments are the following.

–LNO:ps1=N, ps2=N, ps3=N, ps4=N
This option specifies the number of bytes in a page, with **N** as positive integer. The default for **N** depends on your system hardware.

–LNO:tlb1=N, tlb2=N, tlb3=N, tlb4=N
This option specifies the number of entries in the TLB for this cache level, with **N** as a positive integer. The default for **N** depends on your system hardware.

–LNO:tlbcmp1=N, tlbcmp2=N, tlbcmp3=N, tlbcmp4=N, tlbdmp1=N, tlbdmp2=N, tlbdmp3=N, tbldmp4=N
This option specifies the number of processor cycles it takes to service a clean TLB miss (the **tlbcmpx=** options) or a dirty TLB miss (the **tlbdmpx=** options), with **N** as a positive integer. The default for **N** depends on your system hardware.

Following are **LNO Prefetch Options.** These arguments control the prefetch operation.

–LNO:assume_unknown_trip_count={0,1000}
This flag is equivalent to **–LNO:trip_count={0,1000}** and indicates the loop trip count to assume in the absence of feedback. This information is used to avoid prefetches inside LNO. Default value is **1000**.

–LNO:pf1=(ON|OFF), pf2=(ON|OFF), pf3=(ON|OFF), pf4=(ON|OFF)
This options selectively disables or enables prefetching for cache level **x**, for **pfx=(ON|OFF)**

–LNO:prefetch=(0|1|2|3)
This option specifies the levels of prefetching. The options can be one of the following:

0 = Prefetch disabled.

1 = Prefetch is done only for arrays that are always referenced in each iteration of a loop. This is the default.

2 = Prefetch is done without the above restriction.

3 = Most aggressive.

−LNO:prefetch_ahead=N
This option prefetches the specified number of cache lines ahead of the reference. Specify a positive integer for **N**; default is **2**.

−LNO:prefetch_manual=(ON|OFF)
This option specifies whether manual prefetches (through directives) should be respected or ignored.

prefetch_manual=OFF ignores directives for prefetches.

prefetch_manual=ON respects directives for prefetches. This is the default.

−M Run cpp and print list of make dependencies.

−m32 Compile for 32-bit ABI, also known as x86 or IA32. See -m64 for defaults.

−m3dnow
Enable use of 3DNow instructions. The default is **OFF**.

−m64 Compile for 64-bit ABI, also known as AMD64, x86_64, or IA32e. On a 32-bit host, the default is 32-bit ABI. On a 64-bit host, the default is 64-bit ABI if the target platform (-march/-mcpu/-mtune) is 64-bit; otherwise the default is 32-bit.

−macro-expand
Enable macro expansion in preprocessed Fortran source files throughout each file. Without this option specified, macro expansion is limited to preprocessor **#** directives in files processed by the Fortran preprocessor. When this option is specified, macro expansion occurs throughout the source file.

−march=(opteron|athlon|athlon64|athlon64fx|em64t|pentium4|xeon|anyx86|auto)
Compiler will optimize code for the selected platform. **auto** means to optimize for the platform that the compiler is running on, which the compiler determines by reading /proc/cpuinfo. **anyx86** means a generic x86 processor. Under 32-bit ABI, anyx86 is a processor without SSE2/SSE3/3DNow! support; under 64-bit ABI it is a processor with SSE2 but without SSE3/3DNow!. The default is opteron.

−mcmodel=(small|medium)
Select the code size model to use when generating offsets within object files. Most programs will work with **−mcmodel=small** (using 32−bit pointers), but some need **−mcmodel=medium** (using 32−bit pointers for code and 64−bit pointers for data).

−mcpu=(opteron|athlon|athlon64|athlon64fx|em64t|pentium4|xeon|anyx86|auto)
Compiler will optimize code for the selected platform. **auto** means to optimize for the platform that the compiler is running on, which the compiler determines by reading /proc/cpuinfo. **anyx86** means a generic x86 processor. Under 32-bit ABI, anyx86 is a processor without SSE2/SSE3/3DNow! support; under 64-bit ABI it is a processor with SSE2 but without SSE3/3DNow!. The default is opteron.

−MD Write dependencies to **.d** output file

−MDtarget
Use the following as the target for Make dependencies.

−MDupdate
Update the following file with Make dependencies.

−MF Write dependencies to specified output file.

−MG With **−M** or **−MM**, treat missing header files as generated files.

−MM Output user dependencies of source file.

−MMD Write user dependencies to **.d** output file.

−mno-sse
Disable the use of SSE2/SSE3 instructions. SSE2 cannot be disabled under **-m64** and will result in a warning.

−mno-sse2
Disable the use of SSE2/SSE3 instructions. SSE2 cannot be disabled under **-m64** and will result in a warning.

−mno-sse3
Disable the use of SSE3 instructions.

−module *dir*
Create the ".mod" file corresponding to a "module" statement in the directory *dir* instead of the current working directory. Also, when searching for modules named in "use" statements, examine the directory *dir* before the directories established by **-I***dir* options.

−mp Interpret OpenMP directives to explicitly parallelize regions of code for execution by multiple threads on a multi−processor system. Most OpenMP 2.0 directives are supported by **pathf95**, **pathcc** and **pathCC**. See the *PathScale EKOPath Compiler Suite User Guide* for more information on these directives.

−MP With **−M** or **−MM**, add phony targets for each dependency.

−MQ Same as **−MT**, but quote characters that are special to Make.

−msse2 Enable use of SSE2 instructions. This is the default under both **−m64** and **−m32**.

−msse3 Enable use of SSE3 instructions. Default is **ON** under **−march=em64t**. Otherwise, it is **OFF** by default.

−mtune=(opteron|athlon|athlon64|athlon64fx|em64t|pentium4|xeon|anyx86|auto)
Compiler will optimize code for the selected platform. **auto** means to optimize for the platform that the compiler is running on, which the compiler determines by reading /proc/cpuinfo. **anyx86** means a generic x86 processor. Under 32-bit ABI, anyx86 is a processor without SSE2/SSE3/3DNow! support; under 64-bit ABI it is a processor with SSE2 but without SSE3/3DNow!. The default is opteron.

−MT Change the target of the generated dependency rules.

−mx87-precision=(32|64|80)
Specify the precision of x87 floating-point calculations. The default is **80**-bits.

−nobool Do not allow boolean keywords.

−nocpp (For **Fortran** only) Disable the source preprocessor.

See the **−cpp**, **−E**, and **−ftpp** options for more information on controlling preprocessing.

−nodefaultlibs
Do not use standard system libraries when linking.

−noexpopt
Do not optimize exponentiation operations.

−noextend-source
Restrict Fortran source code lines to columns 1 through 72.

See the **−col***n* and **−extend-source** options for more information on controlling line length.

−nog77mangle
The PathScale Fortran compiler modifies Fortran symbol names by appending an underscore, so a name like "foo" in a source file becomes "foo_" in an object file.

However, if a name in a Fortran source file contains an underscore, the compiler appends a second underscore in the object file, so "foo_bar" becomes "foo_bar__", and "baz_" becomes "baz___".

The **−nog77mangle** option suppresses the addition of this second underscore.

−no-gcc (For **C/C++** only) **−no-gcc** turns off the __GNUC__ and other predefined preprocessor macros.

−noinline
Suppress expansion of inline functions. When this option is specified, copies of inline functions are emitted as static functions in each compilation unit where they are called. It is preferable to use **−INLINE:=OFF** or **−IPA:inline=OFF** if you are using IPA (see **ipa**(1)). One of these options must be specified if you are using IPA.

−no-pathcc
−no−pathcc turns off the __PATHSCALE__ and other predefined preprocessor macros.

−nostartfiles
Do not use standard system startup files when linking.

−nostdinc
Direct the system to skip the standard directory, **/usr/include**, when searching for **#include** files and files named on **INCLUDE** statements.

−nostdinc++
Do not search for header files in the standard directories specific to C++.

−nostdlib
No predefined libraries or startfiles.

−o *outfile*
When this option is used in conjunction with the **−c** option and a single C source file, a relocatable object file named *outfile* is produced. When specified with the **−S** option, the **−o** option is ignored. If **−o** and **−c** are not specified, a file named **a.out** is produced. If specified, writes the executable file to *out_file* rather than to **a.out**.

−O(0|1|2|3|s)
Specify the basic level of optimization desired. The options can be one of the following:

0 Turn off all optimizations.

1 Turn on local optimizations that can be done quickly.

2 Turn on extensive optimization. This is the default. The optimizations at this level are generally conservative, in the sense that they are virtually always beneficial, provide improvements commensurate to the compile time spent to achieve them, and avoid changes which affect such things as floating point accuracy.

3 Turn on aggressive optimization. The optimizations at this level are distinguished from **−O2** by their aggressiveness, generally seeking highest−quality generated code even if it requires extensive compile time. They may include optimizations that are generally beneficial but may hurt performance.

This includes but is not limited to turning on the Loop Nest Optimizer, **−LNO:opt=1**, and setting **−OPT:ro=1:IEEE_arith=2:Olimit=9000:reorg_common=ON**.

s Specify that code size is to be given priority in tradeoffs with execution time.

If no value is specified, **2** is assumed.

−objectlist
Read the following file to get a list of files to be linked.

−Ofast Equivalent to **−O3 −ipa −OPT:Ofast −fno−math−errno −ffast−math**. Use optimizations selected to maximize performance. Although the optimizations are generally safe, they may affect floating point accuracy due to rearrangement of computations.

NOTE: **−Ofast** enables **−ipa** (inter-procedural analysis), which places limitations on how libraries and **.o** files are built.

−openmp

Interpret OpenMP directives to explicitly parallelize regions of code for execution by multiple threads on a multi−processor system. Most OpenMP 2.0 directives are supported by **pathf95**, **pathcc** and **pathCC**. See the *PathScale EKOPath Compiler Suite User Guide* for more information on these directives.

−OPT: . . .

This option group controls miscellaneous optimizations. These options override defaults based on the main optimization level.

−OPT:alias=<name>

Specify the pointer aliasing model to be used. By specifying one or more of the following for <*name*>, the compiler is able to make assumptions throughout the compilation:

typed Assume that the code adheres to the ANSI/ISO C standard which states that two pointers of different types cannot point to the same location in memory. This is **ON** by default when **−OPT:Ofast** is specified.

restrict Specify that distinct pointers are assumed to point to distinct, non−overlapping objects. This is **OFF** by default.

disjoint Specify that any two pointer expressions are assumed to point to distinct, non−overlapping objects. This is **OFF** by default.

−OPT:align_unsafe=(ON|OFF)

Instruct the vectorizer (invoked at **−O3**) to aggressively perform vectorization by assuming that array parameters are aligned at 128-bit boundaries. The vectorizer will then generate 128-bit aligned load and store instructions, which are faster than their unaligned counterparts. If the assumption is incorrect, the aligned memory accesses will result in run-time segmentation faults. The default is **OFF**.

−OPT:asm_memory=(ON|OFF)

A debugging option to be used when debugging suspected buggy inline assembly. If **ON**, the compiler assumes each asm has "memory" specified even if it is not there. The default is **OFF**.

−OPT:bb=N

This specifies the maximum number of instructions a basic block (straight line sequence of instructions with no control flow) can contain in the code generator's program representation. Increasing this value can improve the quality of optimizations that are applied at the basic block level, but can increase compilation time in programs that exhibit such large basic blocks. The default is 1300. If compilation time is an issue, use a smaller value.

−OPT:cis=(ON|OFF)

Convert SIN/COS pairs using the same argument to a single call calculating both values at once. The default is **ON**.

−OPT:div_split=(ON|OFF)

Enable or disable changing x/y into x*(recip(y)). This is **OFF** by default, but enabled by **−OPT:Ofast** or **−OPT:IEEE_arithmetic=3**. This transformation generates fairly accurate code.

−OPT:early_mp=(ON|OFF)

This flag has any effect only under **−mp** compilation. It controls whether the transformation of code to run under multiple threads should take place before or after the loop nest optimization (LNO) phase in the compilation process. The default is **OFF**, when the transformation occurs after LNO. Some OpenMP programs can yield better performance by enabling **−OPT:early_mp** because LNO can sometimes generate more appropriate loop transformation when working on the multi-threaded forms of the loops. If **−apo** is specified, the transformation of code to run under multiple threads can only take place after the LNO phase, in which case this flag is ignored.

−OPT:early_intrinsics=(ON|OFF)

When **ON**, this option causes calls to intrinsics to be expanded to inline code early in the backend compilation. This may enable more vectorization opportunities if vector forms of the expanded operations exist. Default is **OFF**.

−OPT:fast_bit_intrinsics=(ON|OFF)
Setting this to **ON** will turn off the check for the bit count being within range for Fortran intrinsics (like BTEST and ISHFT). The default setting is **OFF**.

−OPT:fast_complex=(ON|OFF)
Setting **fast_complex=ON** enables fast calculations for values declared to be of the type *complex*. When this is set to **ON**, complex absolute value (norm) and complex division use fast algorithms that overflow for an operand (the divisor, in the case of division) that has an absolute value that is larger than the square root of the largest representable floating-point number. This would also apply to an underflow for a value that is smaller than the square root of the smallest representable floating point number. **OFF** is the default. **fast_complex=ON** is enabled if **−OPT:roundoff=3** is in effect.

−OPT:fast_exp=(ON|OFF)
This option enables optimization of exponentiation by replacing the runtime call for exponentiation by multiplication and/or square root operations for certain compile-time constant exponents (integers and halfs). This can produce differently rounded results that those from the runtime function. **fast_exp** is **OFF** unless **−O3** or **−Ofast** are specified, or **−OPT:roundoff=1** is in effect.

−OPT:fast_io=(ON|OFF)
(For **C/C++** only) This option enables inlining of printf(), fprintf(), sprintf(),scanf(), fscanf(), sscanf(), and printw(). **−OPT:fast_io** is only in effect when the candidates for inlining are marked as intrinsic to the stdio.h and curses.h files. Default is **OFF**.

−OPT:fast_math=(ON|OFF)
Setting this to **ON** will tell the compiler to use the fast math functions tuned for the processor. The affected math functions include log, exp, sin, cos, sincos, expf and pow. The default setting is **OFF**. It is turned on automatically when **-OPT:roundoff** is at 2 or above.

−OPT:fast_nint=(ON|OFF)
This option uses hardware features to implement NINT and ANINT (both single- and double-precision versions). Default is **OFF** but **fast_nint=ON** is enabled by default if **−OPT:roundoff=3** is in effect.

−OPT:fast_sqrt=(ON|OFF)
This option calculates square roots using the identity sqrt(x)=x*rsqrt(x), where rsqrt is the reciprocal square root operation. This transformation generates fairly accurate code. Default is **OFF**.

−OPT:fast_stdlib=(ON|OFF)
This option controls the generation of calls to faster versions of some standard library functions. Default is **ON**.

−OPT:fast_trunc=(ON|OFF)
This option inlines the NINT, ANINT, and AMOD Fortran intrinsics, both single- and double-precision versions. Default is **OFF**. **fast_trunc** is enabled automatically if **−OPT:roundoff=1** or greater is in effect.

−OPT:fold_reassociate=(ON|OFF)
This option allows optimizations involving reassociation of floating point quantities. Default is **OFF**. **fold_reassociate=ON** is enabled automatically when **−OPT:roundoff=2** or greater is in effect.

−OPT:fold_unsafe_relops=(ON|OFF)
This option folds relational operators in the presence of possible integer overflow. The default is **ON** for **−O3** and **OFF** otherwise.

−OPT:fold_unsigned_relops=(ON|OFF)
This option folds unsigned relational operators in the presence of possible integer overflow. Default is **OFF** .

−OPT:goto=(ON|OFF)
Disable or enable the conversion of GOTOs into higher-level structures like FOR loops. The default is **ON** for **−O2** or higher.

−OPT:IEEE_arithmetic,IEEE_arith=(1|2|3)

Specify the level of conformance to IEEE 754 floating pointing roundoff/overflow behavior. Note that **−OPT:IEEE_a** is a valid abbreviation for this flag. The options can be one of the following:

1 Adhere to IEEE accuracy. This is the default when optimization levels **−O0**, **−O1** and **−O2** are in effect.

2 May produce inexact result not conforming to IEEE 754. This is the default when **−O3** is in effect.

3 All mathematically valid transformations are allowed.

−OPT:IEEE_NaN_Inf=(ON|OFF)

−OPT:IEEE_NaN_inf=ON forces all operations that might have IEEE-754 NaN or infinity operands to yield results that conform to ANSI/IEEE 754-1985, the IEEE Standard for Binary Floating−point Arithmetic, which describes a standard for NaN and inf operands. Default is **ON**.

−OPT:IEEE_NaN_inf=OFF produces non-IEEE results for various operations. For example, **x=x** is treated as **TRUE** without executing a test and **x/x** is simplified to **1** without dividing. **OFF** can enable many common optimizations that can help performance.

−OPT:inline_intrinsics=(ON|OFF)

When **OFF**, this option turns all Fortran intrinsics that have a library function into a call to that function. Default is **ON**.

−OPT:Ofast

Use optimizations selected to maximize performance. Although the optimizations are generally safe, they may affect floating point accuracy due to rearrangement of computations. This effectively turns on the following optimizations: **−OPT:ro=2:Olimit=0:div_split=ON:alias=typed**.

−OPT:Olimit=N

Disable optimization when size of program unit is > **N**. When **N** is 0, program unit size is ignored and optimization process will not be disabled due to compile time limit. The default is **0** when **−OPT:Ofast** is specified, **9000** when **−O3** is specified; otherwise the default is **6000**.

−OPT:pad_common=(ON|OFF)

This option reorganizes common blocks to improve the cache behavior of accesses to members of the common block. This may involve adding padding between members and/or breaking a common block into a collection of blocks. Default is **OFF**.

This option should not be used unless the common block definitions (including EQUIVALENCE) are consistent among all sources making up a program. In addition, **pad_common=ON** should not be specified if common blocks are initialized with DATA statements. If specified, **pad_common=ON** must be used for all of the source files in the program.

−OPT:recip=(ON|OFF)

This option specifies that faster, but potentially less accurate, reciprocal operations should be performed. Default is **OFF**.

−OPT:reorg_common=(ON|OFF)

This option reorganizes common blocks to improve the cache behavior of accesses to members of the common block. The reorganization is done only if the compiler detects that it is safe to do so.

reorg_common=ON is enabled when **−O3** is in effect and when all of the files that reference the common block are compiled at **−O3**.

reorg_common=OFF is set when the file that contains the common block is compiled at **−O2** or below.

−OPT:roundoff=(0|1|2|3) or **−OPT:ro=(0|1|2|3)**

Specify the level of acceptable departure from source language floating−point, round−off, and overflow semantics. The options can be one of the following:

0 = Inhibit optimizations that might affect the floating−point behavior. This is the default when optimization levels **−O0**, **−O1**, and **−O2** are in effect.

1 = Allow simple transformations that might cause limited round−off or overflow differences. Compounding such transformations could have more extensive effects. This is the default when **−O3** is in effect.

2 = Allow more extensive transformations, such as the reordering of reduction loops. This is the default level when **−OPT:Ofast** is specified.

3 = Enable any mathematically valid transformation.

−OPT:rsqrt=(0|1|2)
This option specifies if the RSQRT machine instruction should be used to calculate reciprocal square root. RSQRT is faster but potentially less accurate than the regular square root operation. 0 means not to use RSQRT. 1 means to use RSQRT followed by instructions to refine the result. 2 means to use RSQRT by itself. Default is **1** when -OPT:roundoff=2 or greater, else the default is **0**.

−OPT:space=(ON|OFF) When **ON**, this option specifies that code size is to be given priority in tradeoffs with execution time in optimization choices. Default is **OFF**. This can be turned on either directly or by compiling with **-Os**.

−OPT:transform_to_memlib=(ON|OFF)
When **ON**, this option enables transformation of loop constructs to calls to **memcpy** or **memset**. Default is **ON**.

−OPT:treeheight=(ON|OFF)
The value **ON** enables re−association in expressions to reduce the expressions' tree height. The default is **OFF**.

−OPT:unroll_analysis=(ON|OFF)
The default value of **ON** lets the compiler analyze the content of the loop to determine the best unrolling parameters, instead of strictly adhering to the **−OPT:unroll_times_max** and **−OPT:unroll_size** parameters.

−OPT:unroll_analysis=ON can have the negative effect of unrolling loops less than the upper limit dictated by the **−OPT:unroll_times_max** and **−OPT:unroll_size** specifications.

−OPT:unroll_times_max=N
Unroll inner loops by a maximum of **N**. The default is **4**.

−OPT:unroll_size=N
Set the ceiling of maximum number of instructions for an unrolled inner loop. If **N=0**, the ceiling is disregarded. The default is **40**.

−OPT:wrap_around_unsafe_opt=(ON|OFF)
−OPT:wrap_around_unsafe_opt=OFF disables both the induction variable replacement and linear function test replacement optimizations. By default these optimizations are enabled at **−O3**. This option is disabled by default at **−O0**.

Setting **−OPT:wrap_around_unsafe_opt** to **OFF** can degrade performance. It is provided as a diagnostic tool.

−P Run only the source preprocessor and puts the results for each source file (that is, for *file*.**f**[**90**], *file*.**F**[**90**], and/or *file*.**s**) in a corresponding *file*.**i**. The *file*.**i** that is generated does not contain **#** lines.

−pad-char-literals
(For **Fortran** only) Blank pad all character literal constants that are shorter than the size of the default integer type and that are passed as actual arguments. The padding extends the length to the size of the default integer type.

−pathcc Define __PATHCC__ and other macros.

−pedantic-errors
Issue warnings needed by strict compliance to ANSI C.

−pg Generate extra code to profile information suitable for the analysis program **pathprof**(1). You must use this option when compiling the source files you want data about, and you must also use it when linking. This option turns on application level profiling but not library level profiling (see also -profile). See the gcc man pages for more information.

−profile Generate extra code to profile information suitable for the analysis program **pathprof**(1). You must use this option when compiling the source files you want data about, and you must also use it when linking. This option turns on application level and library level profiling (see also -pg).

−r Produce a relocatable **.o** and stop.

−r*real_spec*
(For **Fortran** only) Specify the default kind specification for real values.

Option	Kind value
−r4	Use **REAL(KIND=4)** and **COMPLEX(KIND=4)** for real and complex variables, respectively (the default).
−r8	Use **REAL(KIND=8)** and **COMPLEX(KIND=8)** for real and complex variables, respectively.

−S Generate an assembly file, *file***.s**, rather than an object file (*file***.o).**

−shared DSO−shared PIC code.

−shared−libgcc
Force the use of the shared libgcc library.

−show Print the passes as they execute with their arguments and their input and output files.

−show-defaults
Show the default options in the **compiler.defaults**(5) file.

−show0 Show what phases would be called, but don't invoke anything.

−showt Show time taken by each phase.

−static Suppress dynamic linking at runtime for shared libraries; use static linking instead.

−static−data
Statically allocate all local variables. Statically allocated local variables are initialized to zero and exist for the life of the program. This option can be useful when porting programs from older systems in which all variables are statically allocated.

When compiling with the **−static−data** option, global data is allocated as part of the compiled object (*file***.o**) file. The total size of any *file***.o** cannot exceed 2 GB, but the total size of a program loaded from multiple **.o** files can exceed 2 GB. An individual common block cannot exceed 2 GB, but you can declare multiple common blocks each having that size.

If a parallel loop in a multi-processed program calls an external routine, that external routine cannot be compiled with the **−static−data** option. You can mix static and multi-processed object files in the same executable, but a static routine cannot be called from within a parallel region.

−std=c++98
-std option for g++.

−std=c89
-std option for gcc/g++.

−std=c99
-std option for gcc/g++.

−std=c9x
-std option for gcc/g++.

−std=gnu++98
-std option for g++.

−std=gnu89
-std option for gcc/g++.

−std=gnu99
-std option for gcc/g++.

−std=gnu9x
-std option for gcc/g++.

−std=iso9899:1990
-std option for gcc/g++.

−std=iso9899:199409
-std option for gcc/g++.

−std=iso9899:1999
-std option for gcc/g++.

−std=iso9899:199x
-std option for gcc/g++.

−stdinc Predefined include search path list.

−subverbose
Produce diagnostic output about the subscription management for the compiler.

−TENV: . . .
This option specifies the target environment option group. These options control the target environment assumed and/or produced by the compiler.

−TENV:frame_pointer=(ON|OFF)
Default is **ON** for C++ and **OFF** otherwise. Local variables in the function stack frame are addressed via the frame pointer register. Ordinarily, the compiler will replace this use of frame pointer by addressing local variables via the stack pointer when it determines that the stack pointer is fixed throughout the function invocation. This frees up the frame pointer for other purposes. Turning this flag on forces the compiler to use the frame pointer to address local variables. This flag defaults to **ON** for C++ because the exception handling mechanism relies on the frame pointer register being used to address local variables. This flag can be turned **OFF** for C++ for programs that do not throw exceptions.

−TENV:X=(0..4)
Specify the level of enabled exceptions that will be assumed for purposes of performing speculative code motion (default is level 1 at all optimization levels) In general, an instruction will not be speculated (i.e. moved above a branch by the optimizer) unless any exceptions it might cause are disabled by this option.

Level **0** - No speculative code motion may be performed.

Level **1** - Safe speculative code motion may be performed, with IEEE−754 underflow and inexact exceptions disabled.

Level **2** - All IEEE−754 exceptions are disabled except divide by zero.

Level **3** - All IEEE−754 exceptions are disabled including divide by zero.

Level **4** - Memory exceptions may be disabled or ignored.

−TENV:simd_imask=(ON|OFF)
Default is **ON**. Turning it **OFF** unmasks SIMD floating-point invalid-operation exception.

−TENV:simd_dmask=(ON|OFF)
Default is **ON**. Turning it **OFF** unmasks SIMD fbating-point denormalized-operand exception.

−TENV:simd_zmask=(ON|OFF)
Default is **ON**. Turning it **OFF** unmasks SIMD fbating-point zero-divide exception.

−TENV:simd_omask=(ON|OFF)
Default is **ON**. Turning it **OFF** unmasks SIMD fbating-point overfbw exception.

−TENV:simd_umask=(ON|OFF)
Default is **ON**. Turning it **OFF** unmasks SIMD fbating-point underfbw exception.

−TENV:simd_pmask=(ON|OFF)
Default is **ON**. Turning it **OFF** unmasks SIMD fbating-point precision exception.

−traditional
Attempt to support traditional K&R style C.

−trapuv Trap uninitialized variables. Initialize variables to the value NaN, which helps your program crash if it uses uninitialized variables. Affects local scalar and array variables and memory returned by alloca(). Does not affect the behavior of globals, malloc()ed memory, or Fortran common data. This option is not supported under 32-bit ABI without SSE2.

−U *name*
Remove any initial defi nition of *name*.

−U*var* Undefi ne a variable for the source preprocessor. See the **−D***var* option for information on defi ning variables.

−u*var* Make the default type of a variable undefi ned, rather than using default Fortran 90 rules.

−v Print (on standard error output) the commands executed to run the stages of compilation. Also print the version number of the compiler driver program and of the preprocessor and the compiler proper.

−version
Write compiler release version information to **stdout**. No input fi le needs to be specifi ed when this option is used.

−W*c,arg1***[,***arg2***...]**

Pass the argument(s) *argi* to the compiler pass *c* where *c* is one of [**pfi bal**]. The **c** selects the compiler pass according to the following table:

Character	**Name**
p	preprocessor
f	front-end
i	inliner
b	backend
a	assembler
l	loader

Sets of these phase names can be used to select any combination of phases. For example, **−Wba,−o,foo** passes the option **−o foo** to the **b** and **a** phases.

−Wall Enable most warning messages.

−WB,: −WB,<arg> passes <arg> to the backend via ipacom.

−Wdeclaration-after-statement
(For **C/C++** only) Warn about declarations after statements (pre-C99).

−Werror-implicit-function-declaration
(For **C/C++** only) Give an error when a function is used before being declared.

−W[no-]aggregate−return
(For **C/C++** only) **−Waggregate−return** warns about returning structures, unions or arrays. **−Wno-aggregate-return** will not warn about returning structures, unions, or arrays.

−W[no-]bad-function-cast
−Wbad−function-cast attempts to support writable-strings K&R style C. **−Wno−bad−function-cast** tells the compiler not to warn when a function call is cast to a non-matching type.

−W[no-]cast-align
(For **C/C++** only) **−Wcast−align** warns about pointer casts that increase alignment. **−Wno−cast−align** instructs the compiler not warn about pointer casts that increase alignment.

−Wno-cast-qual
(For **C/C++** only) **−Wcast-qual** warns about casts that discard qualifi ers. **−Wno−cast-qual** tells the compiler not to warn about casts that discard qualifi ers.

−W[no-]char-subscripts
(For **C/C++** only) **−Wchar−subscripts** warns about subscripts whose type is 'char'. The **−Wno−char−subscripts** option tells the compiler not warn about subscripts whose type is 'char'.

−W[no-]comment
(For **C/C++** only) **−Wcomment** warns if nested comments are detected. **−Wno−comment** tell the compiler not to warn if nested comments are detected.

−W[no−]conversion
(For **C/C++** only) **−Wconversion** warns about possibly confusing type conversions. **−Wno−conversion** tells the compiler not to warn about possibly confusing type conversions.

−W[no−]deprecated
−Wdeprecated will announce deprecation of compiler features. **−Wno−deprecated** tells the compiler not to announce deprecation of compiler features.

−Wno−deprecated-declarations
Do not warn about deprecated declarations in code.

−W[no−]disabled-optimization
−Wdisabled-optimization warns if a requested optimization pass is disabled. **−Wno−disabled-optimization** tells the compiler not warn if a requested optimization pass is disabled.

−W[no−]div−by−zero
−Wdiv-by-zero warns about compile−time integer division by zero. **−Wno−div-by-zero** suppresses warnings about compile-time integer division by zero.

−W[no−]endif−labels
−Wendif-labels warns if #if or #endif is followed by text. **−Wno−endif-labels** tells the compiler not to warn if #if or #endif is followed by text.

−W[no−]error
−Werror makes all warnings into errors. **−Wno−error** tells the compiler not to make all warnings into errors.

−W[no−]fbat−equal
−Wfbat-equal warns if fbating point values are compared for equality. **−Wno−fbat-equal** tells the compiler not to warn if fbating point values are compared for equality.

−W[no−]format
(For **C/C++** only) **−Wformat** warns about printf format anomalies. **−Wno−format** tells the compiler not to warn about printf format anomalies.

−Wno−format−extra−args
(For **C/C++** only) Do not warn about extra arguments to printf-like functions.

−W[no−]format−nonliteral
(For **C/C++** only) With the **−Wformat−nonliteral** option, and if **−Wformat**, warn if format string is not a string literal. For **−Wno−format−nonliteral** do not warn if format string is not a string literal.

−W[no−]format-security
(For **C/C++** only) For **−Wformat-security**, if **−Wformat**, warn on potentially insecure format functions.**−Wfno−format-security**, do not warn on potentially insecure format functions.

−Wno-format-y2k
(For **C/C++** only) Do not warn about 'strftime' formats that yield two-digit years.

−W[no−]id-clash
(For **C/C++** only) **−Wid-clash** warns if two identifiers have the same first <num> chars. **−Wid-clash** tells the compiler not to warn if two identifiers have the same first <num> chars.

−W[no−]implicit
(For **C/C++** only) **−Wimplicit** warns about implicit declarations of functions or variables. **−Wno−implicit** tells the compiler not to warn about implicit declarations of functions or variables.

−W[no−]implicit-function-declaration
(For **C/C++** only) **−Wimplicit-function-declaration** warns when a function is used before being declared. **−Wimplicit-function-declaration** tells the compiler not to warn when a function is used before being declared.

−W[no−]implicit-int
(For **C/C++** only) **−Wimplicit-int** warns when a declaration does not specify a type. **−Wno-implicit-int** tells the compiler not to warn when a declaration does not specify a type.

−W[no−]import
−Wimport warns about the use of the #import directive. **−Wno−import** tells the compiler not to warn about the use of the #import directive.

−W[no−]inline
(For **C/C++** only) **−Winline** warns if a function declared as inline cannot be inlined. **−Wno−inline** tells the compiler not to warn if a function declared as inline cannot be inlined.

−W[no−]larger-than-<*number*>
−Wlarger−than− warns if an object is larger than <number> bytes. **−Wno−larger−than−** tells the compiler not to warn if an object is larger than <number> bytes.

−Wno−long−long
(For **C/C++** only) **−Wlong-long** warns if the long long type is used. **−Wno−long−long** tells the compiler not to warn if the long long type is used.

−W[no−]main
(For **C/C++** only) **−Wmain** warns about suspicious declarations of main. **−Wno−main** tells the compiler not warn about suspicious declarations of main.

−W[no−]missing-braces
(For **C/C++** only) **−Wmissing-braces** warns about possibly missing braces around initializers. **−Wno−missing-braces** tells the compiler not warn about possibly missing braces around initializers.

−W[no−]missing-declarations
(For **C/C++** only) **−Wmissing-declarations** warns about global funcs without previous declarations. **−Wno−missing-declarations** tells the compiler not warn about global funcs without previous declarations.

−W[no−]missing-format-attribute
(For **C/C++** only) For the **−Wmissing-format-attribute** option, if **−Wformat** is used, warn on candidates for 'format' attributes. For **−Wno−missing-format-attribute** do not warn on candidates for 'format'

attributes.

−W[no−]missing-noreturn
(For **C/C++** only) **−Wmissing−noreturn** warns about functions that are candidates for 'noreturn' attribute. **−Wno−missing-noreturn** tells the compiler not to warn about functions that are candidates for 'noreturn' attribute.

−W[no−]missing-prototypes
(For **C/C++** only) **−Wmissing-prototypes** warns about global funcs without prototypes. **−Wno−missing-prototypes** tells the compiler not to warn about global funcs without prototypes.

−W[no−]multichar
(For **C/C++** only) **−Wmultichar** warns if a multi-character constant is used. **−Wno−multichar** tells the compiler not to warn if a multi-character constant is used.

−W[no−]nested-externs
(For **C/C++** only) **−Wnested-externs** warns about externs not at file scope level. **−Wno−nested-externs** tells the compiler not to warn about externs not at file scope level.

−Wno-non-template-friend
(For **C++** only) Do not warn about friend functions declared in templates.

−W[no-]non-virtual-dtor
(For **C++** only) **−Wnon-virtual-dtor** will warn when a class declares a dtor (destructor) that should be virtual.**−Wno-non-virtual-dtor** tells the compiler not to warn when a class declares a dtor that should be virtual.

−W[no-]old-style-cast
(For **C/C++** only) **−Wold-style-cast** will warn when a C-style cast to a non-void type is used. **−Wno-old-style-cast** tells the compiler not to warn when a C-style cast to a non-void type is used.

−WOPT:
Specifies options that affect the global optimizer are enabled at **−O2** or above.

−WOPT:aggstr=N
This controls the aggressiveness of the strength reduction optimization performed by the scalar optimizer, in which induction expressions within a loop are replaced by temporaries that are incremented together with the loop variable. When strength reduction is overdone, the additional temporaries increase register pressure, resulting in excessive register spills that decrease performance. The value specified must be a positive integer value, which specifies the maximum number of induction expressions that will be strength-reduced across an index variable increment. When set at 0, strength reduction is only performed for non-trivial induction expressions. The default is 11.

−WOPT:const_pre=(ON|OFF)
When **OFF**, disables the placement optimization for loading constants to registers. Default is **ON**.

−WOPT:if_conv=(0|1|2)
Controls the optimization that translates simple IF statements to conditional move instructions in the target CPU. Setting to **0** suppresses this optimization. The value of **1** designates conservative if-conversion, in which the context around the IF statement is used in deciding whether to if-convert. The value of **2** enables aggressive if-conversion by causing it to be performed regardless of the context. The default is **1**.

−WOPT:ivar_pre=(ON|OFF)
When **OFF**, disables the partial redundancy elimination of indirect loads in the program. Default is **ON**.

−WOPT:mem_opnds=(ON|OFF)
Makes the scalar optimizer preserve any memory operands of arithmetic operations so as to help bring about subsumption of memory loads into the operands of arithmetic operations. Load subsumption is the combining of an arithmetic instruction and a memory load into one instruction. Default is **OFF**.

−WOPT:retype_expr=(ON|OFF)
Enables the optimization in the compiler that converts 64-bit address computation to use 32-bit arithmetic as much as possible. Default is **OFF**.

−WOPT:unroll=(0|1|2)
Control the unrolling of innermost loops in the scalar optimizer. Setting to 0 suppresses this unroller. The default is 1, which makes the scalar optimizer unroll only loops that contain IF statements. Setting to 2 makes the unrolling to also apply to loop bodies that are straight line code, which duplicates the unrolling done in the code generator, and is thus unnecessary. The default setting of 1 makes this unrolling complementary to what is done in the code generator. This unrolling is not affected by the unrolling options under the **−OPT** group.

−WOPT:val=(0|1|2)
Control the number of times the value-numbering optimization is performed in the global optimizer, with the default being **1**. This optimization tries to recognize expressions that will compute identical runtime values and changes the program to avoid re−computing them.

−W[no-]overloaded-virtual
(For **C++** only) The **−Woverloaded-virtual** option will warn when a function declaration hides virtual functions. **−Wno-overloaded-virtual** tells the compiler not to warn when a function declaration hides virtual functions.

−W[no−]packed
(For **C/C++** only) **−Wpacked** warns when packed attribute of a struct has no effect. **−Wno−packed** tells the compiler not to warn when packed attribute of a struct has no effect.

−W[no−]padded
(For **C/C++** only) **−Wpadded** warns when padding is included in a struct. **−Wno−padded** tells the compiler not to warn when padding is included in a struct.

−W[no−]parentheses
(For **C/C++** only) **−Wparentheses** warns about possible missing parentheses. **−Wno−parentheses** tells the compiler not to warn about possible missing parentheses.

−Wno-pmf-conversions
(For **C++** only) Do not warn about converting PMFs to plain pointers.

−W[no−]pointer-arith
(For **C/C++** only) **−Wpointer-arith** warns about function pointer arithmetic. **−Wno−pointer-arith** tells the compiler not to warn about function pointer arithmetic.

−W[no−]redundant-decls
(For **C/C++** only) **−Wredundant−decls** warns about multiple declarations of the same object. **−Wno−redundant−decls** tells the compiler not to warn about multiple declarations of the same object.

−W[no-]reorder
(For **C/C++** only) The **−Wreorder** option warns when reordering member initializers. **−Wno-reorder** tells the compiler not to warn when reordering member initializers.

−W[no−]return-type
(For **C/C++** only) **−Wreturn−type** warns when a function return type defaults to int. **−Wno−return−type** tells the compiler not to warn when a function return type defaults to int.

−W[no−]sequence-point
(For **C/C++** only) **−Wsequence−point** warns about code violating sequence point rules. **−Wno−sequence−point** tells the compiler not to warn about code violating sequence point rules.

−W[no−]shadow
(For **C/C++** only) **−Wshadow** warns when one local variable shadows another. **−Wno−shadow** tells the compiler not to warn when one local variable shadows another.

−W[no−]sign-compare
(For **C/C++** only) **−Wsign−compare** warns about signed/unsigned comparisons. **−Wsign−compare** tells the compiler not to warn about signed/unsigned comparisons.

−W[no-]sign-promo
(For **C/C++** only) The **−Wsign−promo** option warns when overload resolution promotes from unsigned to signed. **−Wno−sign−promo** tells the compiler not to warn when overload resolution promotes from unsigned to signed.

−W[no−]strict-aliasing
(For **C/C++** only) **−Wstrict−aliasing** warns about code that breaks strict aliasing rules. **−Wno−strict−aliasing** tells the compiler not to warn about code that breaks strict aliasing rules.

−W[no−]strict−prototypes
(For **C/C++** only) **−Wstrict−prototypes** warns about non−prototyped function decls. **−Wno−strict−prototypes** tells the compiler not to warn about non-prototyped function decls.

−W[no−]switch
(For **C/C++** only) **−Wswitch** warns when a switch statement is incorrectly indexed with an enum. **−Wno−switch** tells the compiler not to warn when a switch statement is incorrectly indexed with an enum.

−W[no−]system-headers
(For **C/C++** only) **−Wsystem−headers** prints warnings for constructs in system header fi les. **−Wno−system−headers** tells the compiler not to print warnings for constructs in system header fi les.

−W[no-]synth
(For **C++** only) The **−Wsynth** option warns about synthesis that is not backward compatible with cfront. **−Wno−synth** tells the compiler not to warn about synthesis that is not backwards compatible with cfront.

−W[no−]traditional
(For **C/C++** only) **−Wtraditional** warns about constructs whose meanings change in ANSI C. **−Wno−traditional** tells the compiler not to warn about constructs whose meanings change in ANSI C.

−W[no−]trigraphs
(For **C/C++** only) **−Wtrigraphs** warns when trigraphs are encountered. **−Wno−trigraphs** tells the compiler not to warn when trigraphs are encountered.

−W[no−]undef
−Wundef warns if an undefi ned identifi er appears in a #if directive. **−Wno−undef** tells the compiler not to warn if an undefi ned identifi er appears in a #if directive.

−W[no−]uninitialized
−Wuninitialized warns about uninitialized automatic variables. Because the analysis to fi nd uninitialized variables is performed in the global optimizer invoked at -O2 or above, this option has no effect at **−O0** and **−O1**. **−Wno−uninitialized** tells the compiler not to warn about uninitialized automatic variables.

−W[no−]unknown-pragmas
−Wunknown−pragmas warns when an unknown #pragma directive is encountered. **−Wno−unknown−pragmas** tells the compiler not to warn when an unknown #pragma directive is encountered.

−W[no−]unreachable-code
−Wunreachable−code warns about code that will never be executed. **−Wno−unreachable−code** tells the compiler not to warn about code that will never be executed.

−W[no−]unused
−Wunused warns when a variable is unused. **−Wno−unused** tells the compiler not to warn when a variable is unused.

−W[no−]unused-function
−Wunused−function warns about unused static and inline functions. **−Wno−unused−function** tells the compiler not to warn about unused static and inline functions.

−W[no−]unused-label
−Wunused−label warns about unused labels. **−Wno−unused−label** tells the compiler not to warn about unused labels.

−W[no−]unused-parameter
−Wunused−parameter warns about unused function parameters. **−Wno−unused−parameter** tells the compiler not to warn about unused function parameters.

−W[no−]unused-value
−Wunused−value warns about statements whose results are not used. **−Wno−unused−value** tells the compiler not to warn about statements whose results are not used.

−W[no−]unused-variable
−Wunused−variable warns about local and static variables that are not used. **−Wunused−variable** tells the compiler not to warn about local and static variables that are not used.

−W[no−]write-strings
−Wwrite−strings marks strings as 'const char*'. **−Wno−write−strings** tells the compiler not to mark strings as 'const char *'.

−Wnonnull
(For **C/C++** only) Warn when passing null to functions requiring non-null pointers.

−Wswitch−default
(For **C/C++** only) Warn when a switch statement has no default.

−Wswitch−enum
(For **C/C++** only) Warn when a switch statement is missing a case for an enum member.

−w Suppress warning messages.

−woff Turn off named warnings

−woffall
Turn off all warnings.

−woffoptions
Turn off warnings about options.

−woff*num*
Specify message numbers to suppress. Examples:

- Specifying **−woff2026** suppresses message number 2026.
- Specifying **−woff2026−2352** suppresses messages 2026 through 2352.
- Specifying **−woff2026−2352,2400−2500** suppresses messages 2026 through 2352 and messages 2400 through 2500.

In the message−level indicator, the message numbers appear after the dash.

−Y*c,path*
Set the *path* in which to find the associated phase, using the same phase names as given in the **−W** option. The following characters can also be specified:

I Specifies where to search for include files

S Specifies where to search for startup files (**crt*.o**)

L Specifies where to search for libraries

−zerouv Set uninitialized variables to zero. Affects local scalar and array variables and memory returned by alloca(). Does not affect the behavior of globals, malloc()ed memory, or Fortran common data.

ENVIRONMENT VARIABLES

F90_BOUNDS_CHECK_ABORT
(Fortran) Set to **YES**, causes the program to abort on the first bounds check violation.

F90_DUMP_MAP
(Fortran) If a segmentation fault occurs, print the current process's memory map before aborting. The memory map describes how the process's address space is allocated. The Fortran runtime will print the address of the segmentation fault; you can examine the memory map to see which mapped area was nearest to the fault address. This can help distinguish between program bugs that involve running out of stack space and null pointer dereferences. The memory map is displayed using the same format as the file /proc/self/maps.

FILENV
The location of the **assign** file. See the **assign** (1) man page for more details.

FTN_SUPPRESS_REPEATS
(Fortran) Output multiple values instead of using the repeat factor, used at runtime.

NLSPATH
(Fortran) Flags for runtime and compile-time messages.

PSC_CFLAGS
(C) Flags to pass to the C compiler, pathcc.

PSC_COMPILER_DEFAULTS_PATH
Specifies a path or colon-separated list of paths, designating where the compiler is to look for the **compiler.defaults**(5) file. If the environment variable is set, the path */opt/pathscale/etc* will not be used. If the file cannot be found, then no defaults file will be used, even if one is present in */opt/pathscale/etc*.

PSC_PROBLEM_REPORT_DIR
Name a directory in which to save problem reports and preprocessed source files, if the compiler encounters an internal error. If not specified, the directory used is **$HOME/.ekopath-bugs**.

PSC_CXXFLAGS
(C++) Flags to pass to the C++ compiler, pathCC.

PSC_FFLAGS
(Fortran) Flags to pass to the Fortran compiler, pathf95.

PSC_GENFLAGS
Generic flags passed to all compilers.

PSC_STACK_LIMIT
(Fortran) Controls the stack size limit the Fortran runtime attempts to use. This string takes the format of a floating-point number, optionally followed by one of the characters "k" (for units of 1024 bytes), "m" (for units of 1048576 bytes), "g" (for units of 1073741824 bytes), or "%" (to specify a percentage of physical memory). If the specifier is following by the string "/cpu", the limit is divided by the number of CPUs the system has. For example, a limit of "1.5g" specifies that the Fortran runtime will use no more than 1.5 gigabytes (GB) of stack. On a system with 2GB of physical memory, a limit of "90%/cpu" will use no more than 0.9GB of stack (2/2*0.90).

PSC_STACK_VERBOSE
(Fortran) If this environment variable is set, the Fortran runtime will print detailed information about how it is computing the stack size limit to use.

Standard OpenMP Runtime Environment Variables
These environment variables can be used with OpenMP in either Fortran or C and C++.

OMP_DYNAMIC
Enables or disables dynamic adjustment of the number of threads available for execution. Default is **FALSE**, since this mechanism is not supported.

OMP_NESTED
Enables or disables nested parallelism. Default is **FALSE**.

OMP_SCHEDULE
This environment variable only applies to DO and PARALLEL_DO directives that have schedule type RUNTIME. Type can be STATIC, DYNAMIC, or GUIDED. Default is **STATIC**, with no chunk size specified.

OMP_NUM_THREADS
Set the number of threads to use during execution. Default is number of CPUs in the machine.

PathScale OpenMP Environment Variables
These environment variables can be used with OpenMP in both Fortran and C and C++, except as indicated.

PSC_OMP_AFFINITY
When **TRUE**, the operating system's affinity mechanism (where available) is used to assign threads to CPUs, otherwise no affinity assignments are made. The default value is **TRUE**.

PSC_OMP_AFFINITY_GLOBAL
This environment variable controls where thread global ID or local ID values are used when assigning threads to CPUs. The default is **TRUE** so that global ID values are used for calculating thread assignments.

PSC_OMP_AFFINITY_MAP
This environment variable allows the mapping from threads to CPUs to be fully specified by the user. It must be set to a list of CPU identifiers separated by commas. The list must contain at least one CPU identifier, and entries in the list beyond the maximum number of threads supported by the implementation (256) are ignored. Each CPU identifier is a decimal number between 0 and one less than the number of CPUs in the system (inclusive).

The implementation generates a mapping table that enumerates the mapping from each thread to CPUs. The CPU identifiers in the **PSC_OMP_AFFINITY_MAP** list are inserted in the mapping table starting at the index for thread 0 and increasing upwards. If the list is shorter than the maximum number of threads, then it is simply repeated over and over again until there is a mapping for each thread. This repeat feature allows short lists to be used to specify repetitive thread mappings for all threads.

PSC_OMP_CPU_STRIDE
This specifies the striding factor used when mapping threads to CPUs. It takes an integer value in the range of 0 to the number of CPUs (inclusive). The default is a stride of **1**, which causes the threads to be linearly mapped to consecutive CPUs. When there are more threads than CPUs the mapping wraps around giving a round-robin allocation of threads to CPUs. The behavior for a stride of 0 is the same as a stride of 1.

PSC_OMP_CPU_OFFSET
This specifies an integer value that is used to offset the CPU assignments for the set of threads. It takes an integer value in the range of 0 to the number of CPUs (inclusive). When a thread is mapped to a CPU, this offset is added onto the CPU number calculated after **PSC_OMP_CPU_STRIDE** has been applied. If the resulting value is greater than the number of CPUs, then the remainder is used from the division of this value by the number of CPUs.

PSC_OMP_GUARD_SIZE
This environment variable specifies the size in bytes of a guard area that is placed below pthread stacks. This guard area is in addition to any guard pages created by your O/S.

PSC_OMP_GUIDED_CHUNK_DIVISOR
The value of **PSC_OMP_GUIDED_CHUNK_DIVISOR** is used to divide down the chunk size assigned by the guided scheduling algorithm.

PSC_OMP_GUIDED_CHUNK_MAX
This is the maximum chunk size that will be used by the loop scheduler for guided scheduling.

PSC_OMP_LOCK_SPIN
This chooses the locking mechanism used by critical sections and OMP locks.

PSC_OMP_SILENT
If you set **PSC_OMP_SILENT** to anything, then warning and debug messages from the libopenmp library are inhibited.

PSC_OMP_STACK_SIZE
(Fortran) Stack size specifi cation follows the syntax in decribed in the *OpenMP in Fortran* section of *PathScale Compiler Suite EKOPath User Guide*.

PSC_OMP_STATIC_FAIR
This determines the default static scheduling policy when no chunk size is specifi ed. It is discused in the *OpenMP in Fortran* section of *PathScale Compiler Suite EKOPath User Guide*.

PSC_OMP_THREAD_SPIN
This takes a numeric value and sets the number of times that the spin loops will spin at user-level before falling back to O/S schedule/reschedule mechanisms.

COPYRIGHT

SEE ALSO

pathcc(1), **pathCC**(1), **pathf95**(1), **compiler.defaults**(5), **pathopt2**(1), **assign** (1), **explain**(1), **fsymlist**(1), **pathscale_intro**(7), **pathdb**(1)

PathScale EKOPath Compiler Suite Install Guide

PathScale EKOPath Compiler Suite User Guide

PathScale EKOPath Compiler Suite Support Guide

PathScale Debugger User Guide

PathScale Subscription Management User Guide

Appendix F

Glossary

The following is a list of terms used in connection with the PathScale EKOPath Compiler Suite.

AMD64 AMD's 64-bit extensions to Intel's IA32 (more commonly known as "x86") architecture.

alias An alternate name used for identification, such as for naming a field or a file.

aliasing Two variables are said to be "aliased" if they potentially are in the same location in memory. This inhibits optimization. A common example in the C language is two pointers; if the compiler cannot prove that they point to different locations, a write through one of the pointers will cause the compiler to believe that the second pointer's target has changed.

assertion A statement in a program that a certain condition is expected to be true at this point. If it is not true when the program runs, execution stops with an output of where the program stopped and what the assertion was that failed.

base Set of standard flags used in SPEC runs with compiler.

bind To link subroutines in a program. Applications are often built with the help of many standard routines or object classes from a library, and large programs may be built as several program modules. Binding links all the pieces together. Symbolic tags are used by the programmer in the program to interface to the routine. At binding time, the tags are converted into actual memory addresses or disk locations. Or (bind) to link any element, tag, identifier or mnemonic with another so that the two are associated in some manner. See *alias* and *linker*.

BSS (Block Started by Symbol) Section in a Fortran output object module that contains all the reserved but unitialized space. It defines its label and the reserved space for a given number of words.

CG Code generation; a pass in the PathScale EKOPath Compiler.

common block A Fortran term for variables shared between compilation units (source files). Common blocks are a Fortran-77 language feature that creates a group of global variables. The PathScale EKOPath compiler does sophisticated padding of common blocks for higher performance when the Inter-Procedural Analysis (IPA) is in use.

constant A constant is a variable with a value known at compile time.

DSO (dynamic shared object) A library that is linked in at runtime. In Linux, the C library (glibc) is commonly dynamically linked in. In Windows, such libraries are called DLLs.

DWARF A debugging file format used by many compilers and debuggers to support source level debugging. It is architecture-independent and applicable to any processor or operating system. It is widely used on Unix, Linux, and other operating systems, as well in stand-alone environments.

EBO The Extended Block Optimization pass in the PathScale EKOPath compiler.

EM64T The Intel ®Extended Memory 64 Technology family of chips.

equivalence A Fortran feature similar to a C/C++ union, in which several variables occupy the same are of memory.

executable The file created by the compiler (and linker) whose contents can be interpreted and run by a computer. The compiler can also create libraries and debugging information from the source code.

feedback A compiler optimization technique in which information from a run of the program is then used by the compiler to generate better code. The PathScale EKOPath Compiler Suite uses feedback information for branches, loop counts, calls, switch statements, and variable values.

flag A command line option for the compiler, usually an option relating to code optimization.

gcov A utility used to determine if a test suite exercises all code paths in a program.

IPA (Inter-Procedural Analysis) A sophisticated compiler technique in which multiple functions and subroutines are optimized together.

IR (Intermediate Representation) A step in compilation where code is linked in an intermediate representation so that inter-procedual analysis and optimization can take place.

linker A utility program that links a compiled or assembled program to a particular environment. Also known as a "link editor," the linker unites references between program modules and libraries of subroutines. Its output is a load module, which is executable code ready to run in the computer.

LNO (loop nest optimizer) Performs transformation on a loop nest, improves data cache performance, improves optimization opportunities in later phases of compiling, vectorizes loops by calling vector intrinsics, parallelizes loops, computes data dependency information for use by code generator, can generate listing of transformed code in source form.

MP Multiprocessor

NUMA Non-uniform memory access is a method of configuring a cluster of microprocessors in a multiprocessing system so that they can share memory locally, improving performance and the ability of the system to be expanded. NUMA is used in a symmetric multiprocessing (SMP) system.

object_file The intermediate representation of code generated by a compiler after it processes a source file.

pathcov The version of `gcov` that PathScale supports with its compilers. Other versions of `gcov` may not work with code generated by the PathScale EKOPath Compiler Suite, and are not supported by PathScale.

pathprof The version of `gprof` that PathScale supports with its compilers. Other versions of `gprof` may not work with code generated by the PathScale EKOPath Compiler Suite, and are not supported by PathScale.

peak Set of optional flags used with compiler in SPEC runs to optimize performance.

SIMD (Single Instruction Multiple Data) An i386/AMD64 instruction set extension which allows the CPU to operate on multiple pieces of data contained in a single, wide register. These extensions were in three parts, named MMX, SSE, and SSE2.

SMP Symmetric multiprocessing is a "tightly-coupled," "share everything" system in which multiple processors working under a single operating system access each other's memory over a common bus or "interconnect" path.

source_file A software program, usually made up of several text files, written in a programming language, that can be converted into machine-readable code through the use of a compiler.

SPEC (Standard Performance Evaluation Corporation) SPEC provides a standardized suite of source code based upon existing applications that has already been ported to a wide variety of platforms by its membership. The benchmarker takes this source code, compiles it for the system in question and tunes the system for the best results. See http://www.spec.org/ for more information.

SSE3 Instruction set extension to Intel's IA_32 and IA_64 architecture to speed processing. These new instructions are supposed to enable and improve hyperthreading rather than floating-point operations.

TLB Translation Look aside Buffer

vectorization An optimization technique that works on multiple pieces of data at once. For example, the PathScale EKOPath Compiler Suite will turn a loop computing the mathematical function `sin()` into a call to the `vsin()` function, which is twice as fast.

WHIRL The intermediate language (IR) used by compilers allowing the C, C++, and Fortran front-ends to share a common backend. It was developed at Silicon Graphics Inc. and is used by the Open64 compilers.

x86_64 The Linux 64-bit application binary interface (ABI).

Index

PathScale™

PATHSCALE, INC.
2071 STIERLIN COURT, STE. 200
MOUNTAIN VIEW, CA 94043 USA

TEL 650.934.8100
FAX 650.428.1969
PATHSCALE.COM

Accelerating Cluster Performance