

OpenUH Compiler Suite

User's Guide

Version alpha

University of Houston

Computer Science Department

High Performance Tools Group

Dr. Barbara Chapman

Last Modified: 1/26/2006

Table of Contents

I. Introduction	3
II. OpenUH Fortran Compiler.....	3
III. OpenUH C Compiler	3
IV. Porting and Compatibility.....	4
V. Tuning Quick Reference	4
VI. Tuning Options	5
VII. Using OpenMP	11
VIII. Debugging and Troubleshooting.....	14
VIV. Appendixes	14

I. Introduction

The OpenUH compiler is an extremely large software program with over 7 million lines of code. OpenUH is based on SGI's open source Pro64 compiler, which targets the IA-64 Linux platform. OpenUH merges work from the two major branches of Open64 (ORC and Pathscale) to exploit all upgrades and bug fixes. OpenUH is a portable OpenMP compiler, which translates *OpenMP 2.0* directives in conjunction with C, C++, and FORTRAN 77/90 (and some FORTRAN 95).

OpenUH is available as stand-alone software or with the Eclipse integrated development environment. It has a full set of optimization modules, including an interprocedural analysis (IPA) module, a loop nest optimizer (LNO), and global optimizer. A variety of state-of-the-art analyses and transformations are available, sometimes at multiple levels.

OpenUH forms part of the official source code tree developer of Open64 (<http://open64.sourceforge.net>), which makes results available to the entire community.

This user guide covers issues related to installation, flag usages, optimization components, and portability of OpenUH.

II. OpenUH Fortran Compiler

1. Using the Fortran compiler

```
$ uhf90 <flags> foo.f
```

Note: OpenUH currently does not support Fortran programs on Pentium systems. Fortran programs are supported on the Itanium systems.

III. OpenUH C Compiler

1. Using the C compiler

```
$ uhcc <flags> foo.c
```

On Pentium systems, users must use the source-to-source translator to compile C programs. After compilation, files containing the translated source code are generated. The generated files must be recompiled and linked using GCC to create an executable.

To compile using the source-to-source translator:

```
$uhcc -clist <foo.c>
```

This will create the files: `foo.w2c.c` and `foo.w2c.h`, which contain the optimized/translated source code. To compile this code:

```
$gcc foo.w2c.c -I <path>
```

The <path> directory should contain whirl2c.h. This file is available as part of OpenUH's distribution.

Note: The source-to-source translation feature for the current release of OpenUH is functional only for C programs. Also note that the current release of the source-to-source translator is a beta test release. Development of a more stable and complete source-to-source translator is currently underway and will be available in future releases of OpenUH.

IV. Porting and Compatibility

OpenUH currently runs on Itanium I and II systems and Pentium systems running different versions of LINUX.

V. Tuning Quick Reference

1. Basic Optimizations

Basic optimizations provided by OpenUH

Optimization Flags	Definition
CG::	option group to control code generation
O2 (Optimization Level 2)	whole program optimizations
-O3 (Optimization Level 3)	includes Loop nest optimizations.
CLIST::	option group to control C source listing of compiler intermediate
DEBUG::	option group to debugging options
DEFAULT::	default options (abi,isa,proc)
GRA::	option group to control global register allocation
INLINE::	specify inline processing option group
INTERNAL::	option group to control features while testing
IPA::	option group to control interprocedural optimizations
LANG::	option group to control language features
LIST::	option group to control listing file and contents
LMSG::	option group to control error/warning messages in ld
LNO::	option group to control loop nest optimization
MP::	option group to control distributed shared memory optimization
OPT::	option group to control optimization
PROMP::	option group to generate ProMP analysis file
SWP::	option group to control software pipelining

TARG::	option group to control compilation target
TENV::	option group to control target environment
VHO::	option group to control who lowering
WOPT::	option group internal
help::	print list of possible options that contain given string

2. Interprocedural Analysis (IPA)

The optimizations based on interprocedural analysis include procedure inlining, intrinsic function inlining, cloning for constants, dead function and dead variable elimination and interprocedural constant propagation. The Inter Procedural Analyzer (IPA) works on the entire program, across procedure and file boundaries.

Example:

```
$uhcc -ipa foo.c.
```

3. Performance Analysis

OpenUH features a performance analysis tool called Dragon. Dragon is a tool for browsing application source code, callgraph, flowgraph, profiling information and more.

Compile commands:

```
$<compiler name> -c -ipa -O2 -dragon foo.f
```

e.g. \$uhf90 -c -ipa -O2 -dragon foo.f

Link commands:

```
$<compiler name> -ipa -O2 -dragon foo1.o foo2.o program_name
```

e.g. \$uhf90 -ipa -O2 -dragon foo1.o foo2.o program_name

Add the `-mp` flag into the compile and link commands for OpenMP programs. Additional user support for Dragon is provided in the Dragon User's guide (www.cs.uh.edu/~dragon).

VI. Tuning Options

1. Basic Optimizations: -O flag

-O2 (Optimization Level 2)

Whole program optimizations.
-O3 (Optimization Level 3)
Includes Loop nest optimizations.

2. Inter-procedural Analysis (IPA)

In the Open64 Compiler, the Interprocedural Analysis phase occurs with the addition of the `-IPA` flag in the command line. For example: `uhcc -ipa foo.c`. The major analysis and optimizations in IPA include the following:

- Inlining
- Padding analysis
- Construction of the callgraph
- Global variable optimization
- Dead function elimination
- Simple alias analysis
- Cloning analysis
- Constant propagation
- Function cloning
- Array section analysis
- Common block padding/splitting
- Procedure reordering

See appendix A for a detailed list of IPA flag options and corresponding definitions.

3. Loop Nest Optimization (LNO)

The LNO component of OpenUH is based on a unified cost model and a model of the target cache. The Loop Nest Optimizer (LNO) optimizes the memory performance of the program's loop nests. It performs the following optimizations:

- locality optimization
- auto parallelization
- array privatization
- heuristics integrated with software pipelining that employ LNO
- dependency analysis
- register/cache blocking (tiling)
- array padding
- software pipeline

LNO includes well-known uni-modular loop transformations such as:

- Loop Peeling
- Loop Tiling

- Vector Data Pre-fetching
- Loop Fission
- Loop Fusion
- Loop Unroll and Jam
- Loop Interchange

For example, to turn on loop nest optimizations, issue the following command:

```
$uhcc -LNO:opt=1 foobar.c
```

The LNO module also performs source-to-source translation. The source-to-source translator can take as input a program written with OpenMP constructs and explicitly creates parallelism in the code that is OpenMP independent.

To use the loop nest optimizations source-to-source translator:

```
$uhcc -clist -O3 -LNO:<options> foo.c
```

Example with flag options set:

```
$uhcc -clist -O3 -LNO:blocking_size=4:interchange=OFF <foo.c>
```

See additional instructions for using the source-to-source translator in the above section entitled OpenUH C Compiler.

Use the following command to view transformations made by LNO:

```
$uhcc -LIST:cite -O3 -LNO:<options> foo.c
```

This will create the files:

- foo-after-lno.c: (equivalent to foo.w2c.c)
- foo-after-lno.h: (equivalent to foo.w2c.h)
- foo.l: Report of all the transformations done to the source code and cost modeling analysis.
- foo.loc: Source code mapping of the generated code to the original code.

Pragmas/directives can also be inserted into the code to tell the LNO what optimizations to perform. See appendix B for LNO flag options and their default values.

4. Code Generation (CG)

The code generation phases performs IF conversion and software pipelining, and detects recurrences in the code. Hyperblocks are formed, frequency based block

reordering is performed, as are global code motion that moves instructions between basic blocks, and peephole optimizations.

The Code Generator (CG) operates on a representation closer to machine instructions (OPs), different from both WHIRL and from WOPT's hashed SSA form. CG's main phases are:

- CGEXP: Construct CG's CFG, expand WHIRL into OP instructions.
- Various loop transformations, including if-conversion, partial and full unrolling, elimination of redundant loads and stores across iterations, and the software pipeliner (SWP)
- Instruction scheduling, including general code motion (GCM)
- General register allocation (GRA) -- allocation of registers that cross multiple blocks
- Local register allocation (LRA) -- allocation of registers localized to a single block
- More instruction scheduling
- CGEMIT -- Emit assembly code and elf/dwarf info

In addition, two CG phases are each invoked several times intermingled with the other phases:

- Control flow optimization (CFLOW) -- branch optimization, and the merging, reordering, cloning, and removal of basic blocks.
- Extended block optimizer (EBO) -- basically a peephole optimizer that also peeps around corners into nearby blocks

Many of these phases are skipped or truncated if the optimization level is -O0 or -O1. The only option enabled for code generation in OpenUH is gcm:

-CG:gcm=OFF turns off the instruction-level global code motion optimization. The default is ON (ref pathscale user guide).

To invoke the code generator:

```
$uhf90 -cg:gcm=OFF foo.f
```

5. Feedback Directed Optimization (FDO)

The OpenUH compiler has an analysis method that uses dynamic program information. That is, the program can be analyzed using a “feedback file”. A feedback file is an instrumentation file that the programmer can obtain by running the program with the feedback flag and generating an instrumentation file. The feedback file contains program profiling information. When the executable program is run the feedback file is generated. The feedback file is then used as input when the program is compiled again. The sequence of commands to do this for file “foo.c” is as follows:

Compile:

```
$ uhcc -fbcreate fdb -fb_type=1 -fb_phase=0 foo.c
```

Run (this will create a file with name beginning with “fdb”):

```
$ ./a.out
```

Compile again with the feedback file:

```
$ uhcc -fb_opt fdb foo.c
```

Note: The fdb file specified in the above compile command line can be any name. For example if the user specified the file name as “filename” than a feedback file starting with “filename” will generate instead after running the program. Also note that a.out is the default executable generated when no name is specified for the executable when compiling.

Additional flag options for FDO can be found below.

Feedback Flag	Defnition
fb:	Specify feedback file for be
fb_cdir:	Option needed to tell pixie where to emit the Count Files
fb_create:	Option needed for feedback file generation
fb_opt:	Option needed for using feedback files
fb_phase=:	Option needed to tell compile feedback phase
fb_type=:	Option need for the type of profiles
fb_xdir:	Option needed to tell pixie where to emit the pixified dso's
fbexe:	Option needed to specify the name of the pixified binary to prof
fbgen:	Option needed for generating feedback files
Fbranch	
Probabilities:	Use profiling information for branch probabilities
fbuse:	Option needed to use feedback files

6. OPT

Options to control general optimization. For example, to enable early MP lowering and dump c source with nested PU printed at the file level:

```
$uhcc -mp -CLIST:emit_nested_pu=on -OPT:early_mp=on foo.c
```

7. WOPT

The WHIRL Optimizer (WOPT) operates on code at a procedural level. It handles one program unit at a time, and its analysis and code transformation work on the entire procedure (as opposed to the more local optimization performed by the code generator (CG), and to the inter-procedural analysis handled by IPA). In addition to performing optimizing transformations to the code, WOPT computes def-use and alias information for other phases of the compiler.

For historic reasons, WOPT is sometimes called WOPT, OPT, SOPT, PREOPT, the Global Optimizer, and Pre_Optimizer in the code files and documentation.

Phases of WOPT:

Depending on the level of optimization, WOPT may be invoked multiple times on the same program unit during different compiler phases. The phases are enumerated as PREOPT_PHASES in the file `be/opt/optimizer.h`.

WOPT PHASES NAME	USAGE
PREOPT_PHASE	used for PREOPTPHASE
PREOPT_LNO_PHASE	used for LNO phase
PREOPT_DUONLY_PHASE	called by LNO, but will disable optimization
MAINOPT_PHASE	Used when optimization level \geq O2
PREOPT_IPA0_PHASE	called by IPL
PREOPT_IPA1_PHASE	called by main IPA

At -O2 and above, the MAINOPT_PHASE of WOPT is invoked just before CG. During MAINOPT, WOPT performs its full set of optimizations and generates alias information for CG.

At optimization levels higher than -O2, WOPT is also invoked for one or more PREOPT phases. During PREOPT phases WOPT generates def-use and alias info for other parts of the compiler. The various PREOPT phases do not perform the full set of WOPT optimizations. In particular, the SSA partial redundancy elimination (SSAPRE) algorithm is not run.

For example, at -O3 (or -O2 -LNO), the loop nest optimizer (LNO) invokes the PREOPT_LNO_PHASE of WOPT to generate the def-use info needed to construct LNO's dependence graph. If IPA is invoked (often as -O2 -IPA, -O3 -IPA, or -Ofast), then IPA first invokes the PREOPT_IPA1_PHASE of WOPT to generate procedure summary information IPA requires.

VII. Using OpenMP

1. An Overview of OpenMP

OpenMP is a set of extensions to Fortran/C/C++. To compile an OpenMP program, a user needs to specify `-mp` option. Without this option, an OpenMP program will be treated as an ordinary sequential program.

For C and C++, OpenMP pragmas take the form:

- `#pragma omp construct [clause [clause]...]`

For Fortran, OpenMP directives take one of the forms:

- `C$OMP construct [clause [clause]...]`
- `!$OMP construct [clause [clause]...]`
- `*$OMP construct [clause [clause]...]`

OpenMP contains compiler directives, library routines and environment variables. A team of threads are created at the beginning of a parallel region denoted as a *parallel* directive and terminated at the end of a parallel region. Worksharing directives are usually used to share the work among threads: *do* in Fortran or (*for* in C), *sections*, *workshare*, and *single*. The “*for*” construct splits up loop iterations among the threads in a team; Note that there are four kinds of different loop scheduling: *static*, *dynamic*, *guided* and *runtime* with a specified or default chunk size. The “*sections*” work-sharing construct gives a different structured block to each thread. By default, there is a barrier at the end of the “*omp for*” or the “*omp sections*”. The *single* construct denotes a block of code that is executed by only one thread. Use the “*nowait*” clause to turn off the barrier.

There is a list of data sharing attribute clauses: *default*, *shared*, *private*, *firstprivate*, *lastprivate* and *reduction*. One can selectively change storage attributes constructs using the following clauses: *shared*, *private*, *firstprivate*, *lastprivate* and *threadprivate*. The default status can be modified with: `DEFAULT (PRIVATE | SHARED | NONE)`. The *reduction* clause effects the way variables are shared: `reduction (op : list)`. The variables in “*list*” must be shared in the enclosing parallel region. Threads communicate by sharing

variables. Unintended sharing of data can lead to outcome changes as the threads are scheduled differently. Therefore, synchronization directives are needed to protect shared variables and avoid conflicts: *critical*, *barrier*, *atomic*, *flush* and *ordered*. Only one thread at a time can enter a critical section. The *master* construct denotes a structured block that is only executed by the master thread, and the other threads just skip it.

There are several different kinds of OpenMP runtime library routines. Lock routines are as: `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`. Besides, there are some runtime environment routines as following:

To modify/Check the number of threads:

- `omp_set_num_threads()`, `omp_get_num_threads()`, `omp_get_thread_num()`, `omp_get_max_threads()`

To turn on/off nesting and dynamic mode:

- `omp_set_nested()`, `omp_set_dynamic()`, `omp_get_nested()`, `omp_get_dynamic()`

Are we in a parallel region?

- `omp_in_parallel()`

How many processors in the system?

- `omp_num_procs()`

What is the elapsed time?

- `omp_get_wtime()` and `omp_get_wtick()`

There are four OpenMP environment variables: `OMP_SCHEDULE`, `OMP_NUM_THREADS`, `OMP_DYNAMIC` and `OMP_NESTED`. `OMP_SCHEDULE` sets the loop schedule; `OMP_NUM_THREADS` specifies the involved number of threads; `OMP_DYNAMIC` enables the dynamic adjustments of threads in parallel regions; `OMP_NESTED` enables or disables the nested parallelism.

2. Example Codes

We illustrate how to compile and run your code using two examples. The first one is the “hello world” program and the second one is an example using reduction.

2.1 A “Hello World” example

In figure 2-1 below, we display a “hello world” example code in OpenMP C.

```
#include <omp.h>
main () {
int nthreads, tid;
/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel private(nthreads, tid)
{
/* Obtain thread number */
tid = omp_get_thread_num();
printf("Hello World from thread = %d\n", tid);
/* Only master thread does this */
if (tid == 0)
{
nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
} /* All threads join master thread and disband */
}
```

Figure 2-1 A “hello world” example in C.

To specify the number of threads (e.g. 4 threads), type

```
$export OMP_NUM_THREADS=4
```

We may save the code in Figure 2-1 as hello.c. To compile this program, type

```
$uhcc -mp hello.c
```

To run this code, type

```
$/a.out
```

The following is the output of this program:

```
Hello World from thread = 2
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 1
Hello World from thread = 3
```

2.2 An example program with REDUCTION

In Figure 2-2, we show an OpenMP Fortran program with REDUCTION. We save this program as reduction.f90.

```
PROGRAM REDUCTION
INTEGER I, N
REAL A(100), B(100), SUM
! Some initializations
N = 100
DO I = 1, N
  A(I) = I * 1.0
  B(I) = A(I)
ENDDO
SUM = 0.0
!$OMP PARALLEL DO REDUCTION(+:SUM)
DO I = 1, N
  SUM = SUM + (A(I) * B(I))
ENDDO
PRINT *, ' Sum = ', SUM
END
```

Figure 2-2 An OpenMP reduction example.

To compile it, type

```
$uhf90 -mp reduction.f90
```

To run it with 2 threads, type

```
$export OMP_NUM_THREADS=2
$./a.out
```

The following is the output of this program:

```
Sum = 328350.000000
```

VIII. Debugging and Troubleshooting

OpenUH currently provides debugging support through the DDD debugger. The -g flag debugger is currently disabled and will be available in future releases of the compiler.

VIV. Appendixes

Appendix A. IPA flag options

IPA Options	Definitions
-IPA:forcedepth=n	Inline all routines at or below the specified depth in the call tree Performed regardless of space limitations
-IPA:inline=off	Suppress invocation of the inliner Default is ON
-IPA:maxdepth=n	Inline all routines at or below the specified depth in the call tree Performed subject to space limitations
-IPA:min_hotness=n	Specify that a call site's invocation count must exceed a specified number before it can be inlined (applicable during feedback compilation only) Call site's invocation count must exceed n before it can be inlined by IPA
-IPA:multi_clone=n	Specify the maximum number of clones that can be created from a single function
-IPA:node_bloat=n	Specify the maximum percentage growth in the number of procedures relative to the original program during cloning
-IPA:plimit=n	Suppress inlining into a function once a specified size (in blocks) is reached Default is 2500
-IPA:callee_limit=n	Indicate that functions exceeding a specified size limit are never to be automatically inlined Default is 500
-IPA:small_pu=n	Indicate that functions smaller than a specified number of blocks are not subject to the -IPA:plimit restriction Default is 30
-IPA:space=n	Inline until a specified percent factor increase in code size is reached Default is 100; Higher values can be specified for small programs
-IPA:alias=OFF	Disable alias and mod-ref analyses
-IPA:addressing=OFF	Disable address-taken analysis
-IPA:cgi=OFF	Disable constant propagation for global variables
- IPA:common_pad_size=n	Specify the pad size for common block padding
-IPA:cprop=OFF	Disable constant propagation for parameters
-IPA:dfe=OFF	Disable dead function elimination

-IPA:dve=OFF	Disable dead variable elimination
-IPA:linear	Enable linearization of array references
-IPA:pu_reorder	Enable or disable procedure re-ordering optimizations
-IPA:field_reorder	Enable field reordering optimization to minimize data cache misses
-IPA:split=OFF	Disable common block splitting

Appendix B. LNO flag options and default values.

The following text comes from the MAN pages for LNO from SGI.

Suboption Action

auto_dist[= (ON|OFF)]

Distributes local arrays in common blocks that are accessed in parallel. The default is OFF.

This optimization works with either automatic parallelism or parallelism using directives; it is always safe, and does not affect the layout of arrays in virtual space, and does not incur addressing overhead.

fission=n Controls loop fission. n can be one of the following:

- 0 Disables loop fission.
- 1 Performs normal fission as necessary. This is the default.
- 2 Specifies that fission be tried before fusion.

If -LNO:fission=n and -LNO:fusion=n are both set to 1 or to 2, fusion is performed.

fusion=n Controls loop fusion. n can be one of the following:

- 0 Disables loop fusion.
- 1 Performs standard outer loop fusion. This is the default.

- 2 Specifies that outer loops should be fused, even if it means partial fusion.

The compiler attempts fusion before fission. The compiler performs partial fusion if not all levels can be fused in the multiple-level fusion.

If `-LNO=fission=n` and `-LNO:fusion=n` are both set to 1 or to 2, fusion is performed.

`fusion_peeling_limit=n`

Sets the limit for the number of iterations allowed to be peeled in fusion, where $n \geq 0$. By default, $n=5$.

`gather_scatter=n`

Performs gather-scatter optimizations. n can be one of the following:

- 0 Disables all gather-scatter optimization.
- 1 Performs gather-scatter optimizations on non-nested IF statements. This is the default.
- 2 Performs multi-level gather-scatter optimizations.

`ignore_pragmas[= (ON|OFF)]`

Specifies that the command line options override directives in the source file. The default is OFF.

`local_pad_size=n`

Specifies the amount by which to pad local array dimensions. By default, the compiler automatically chooses the amount of padding to improve cache behavior for local array accesses.

`non_blocking_loads[= (ON|OFF)]`

(C/C++ and F77 only) Specifies whether the processor blocks on loads. If not set, the default of the current processor is used.

`oinvar[= (ON|OFF)]`

Controls outer loop hoisting. The default is ON.

`opt=n` Controls the LNO optimization level. n can be one of the following:

- 0 Disables nearly all loop nest optimization.
- 1 Performs full loop nest transformations. This is the default.

outer[= (ON|OFF)]

Enables or disables outer loop fusion. The default is ON.

parallel_overhead=num_cycles

Overrides internal compiler estimates concerning the efficiency to be gained by executing certain loops in parallel rather than serially. num_cycles specifies the number of processor cycles. Specify an integer for num_cycles. The default is 2600.

pure=n (uhcc/uhCC, C/C++)

Tells the compiler how to use the #pragma pure and #pragma no side effects directives when performing parallel analysis. #pragma no side effects may read its arguments and unspecified global data; #pragma pure can read only its arguments; neither directive can modify its arguments or global data. Specify 0, 1, or 2 for n, as follows:

n Value Description

- 0 The compiler ignores the #pragma pure and #pragma no side effects directives when gathering information for parallelization analysis.
- 1 The compiler interprets the #pragma pure and #pragma no side effects directives per their definitions when gathering information for parallelization analysis.
- 2 The compiler interprets #pragma no side effects as #pragma pure when gathering information for parallelization analysis. This option is provided because you may declare a function to have no side effects, when in fact, it is pure, except for references to system variables such as errno. In these cases, you can treat no side effects functions as if they were pure for the purposes of parallelization.

pure=n (Fortran 90)

Specifies the extent to which the compiler should consider the effect of a PURE procedure or a !DIR\$ NOSIDEEFFECTS directive when performing parallel analysis. Specify 0, 1, or 2 for n, as follows:

n	Value	Description
0		Directs the compiler to ignore a PURE attribute and the !DIR\$ NOSIDEEFFECTS directive.
1		Directs the compiler to consider the fact that PURE procedures and procedures preceded by a !DIR\$ NOSIDEEFFECTS directive do not modify global data or procedure arguments when performing parallel analysis. Default.
2		Asserts to the compiler that that PURE procedures and procedures preceded by a !DIR\$ NOSIDEEFFECTS directive do not modify global data, do not modify procedure dummy arguments, and do not access global data.

This setting asserts that the only non-local data items referenced by the procedure are the dummy arguments to the procedure. This is an extension of the Fortran standard meaning of PURE and of the meaning of !DIR\$ NOSIDEEFFECTS. At this setting, more aggressive parallelization can occur if procedures are known not to access global data.

`vintr[= (ON|OFF)]`

Specifies that vectorizable versions of the math intrinsic functions should be used. The default is ON.

The loop transformation arguments allow you to control cache blocking, loop unrolling, and loop interchange. They are as follows:

`blocking[= (ON|OFF)]`

Specify `blocking=OFF` to disable the cache blocking transformation. The default is ON.

`blocking_size=n`

Specifies a block size that the compiler must use when

performing any blocking. Specify a positive integer number that represents the number of iterations.

`interchange[= (ON|OFF)]`

Specifies whether or not loop interchange optimizations are performed. The default is ON.

`ou=n` 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.

`ou_deep[= (ON|OFF)]`

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 it generates much faster code whenever wind-down loop execution costs are important. The default is ON.

`ou_further=n`

Specifies whether or not the compiler performs outer loop unrolling on wind-down loops. Specify an integer for n.

`ou_max=n` Indicates that the compiler can unroll as many as n copies per loop, but no more.

`ou_prod_max=n`

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.

`pwr2[= (ON|OFF)]`

(C/C++ and F77 only) Specifies whether to ignore the leading dimension (set to OFF to ignore).

You can disable additional unrolling by specifying `-LNO:ou_further=999999`. Unrolling is enabled as much as is sensible by specifying `-LNO:ou_further=3`.

Certain arguments allow you to describe the target cache memory system. The numbering in the following arguments starts with the cache level closest to the processor and works outward:

`assoc1=n, assoc2=n, assoc3=n, assoc4=n`

Specifies the cache set associativity. For a fully associative cache, such as main memory, set n to any

sufficiently large number, such as 128. Specify a positive integer for n. Specifying n=0 indicates that there is no cache at that level.

cmp1=n, cmp2=n, cmp3=n, cmp4=n
dmp1=n, dmp2=n, dmp3=n, dmp4=n

Specifies, in processor cycles, the time for a clean miss (cmpx=) or dirty miss (dmpx=) to the next outer level of the memory hierarchy. This number is approximate because it depends upon a clean or dirty line, read or write miss, etc. Specify a positive integer for n. Specifying n=0 indicates that there is no cache at that level.

cs1=n, cs2=n, cs3=n, cs4=n

Specifies the cache size. The value n can be 0, or it can be a positive integer followed by one of the following letters: k, K, m, or M. This specifies the cache size in Kbytes or Mbytes. Specifying 0 indicates that there is no cache at that level.

cs1 refers to the primary cache. cs2 refers to the secondary cache. cs3 refers to memory. cs4 refers to disk. The default cache size for each type of cache depends on your system. You can use the -LIST:options=ON option to see the default cache sizes used during your compilation. In addition you can enter the following command to see the secondary cache size(s) on your system:

```
hinv -c memory | grep Secondary
```

is_mem1[= (ON|OFF)]
is_mem2[= (ON|OFF)]
is_mem3[= (ON|OFF)]
is_mem4[= (ON|OFF)]

Specifies that certain memory hierarchies should be modeled as memory, not cache. The default is OFF for each option.

Blocking can be attempted for this memory hierarchy level, and blocking appropriate for memory, rather than cache, is applied. No prefetching is performed, and any prefetching options are ignored. If an -OPT:is_memx[= (ON|OFF)] option is specified, the corresponding assocx=n specification is ignored, any cmpx=n and dmpx=n options on the command line are ignored.

ls1=n, ls2=n, ls3=n, ls4=n

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 that there is no cache at that level.

Certain arguments control the TLB. The TLB is a cache for the page table, and it is assumed to be fully associative. The TLB control arguments are as follows:

ps1=n, ps2=n, ps3=n, ps4=n

Specifies the number of bytes in a page. Specify a positive integer for n. The default n depends on your system hardware.

tlb1=n, tlb2=n, tlb3=n, tlb4=n

Specifies the number of entries in the TLB for this cache level. Specify a positive integer for n. The default n depends on your system hardware.

tlbcmp1=n, tlbcmp2=n, tlbcmp3=n, tlbcmp4=n

tlbdmp1=n, tlbdmp2=n, tlbdmp3=n, tlbdmp4=n

Specifies the number of processor cycles it takes to service a clean TLB miss (the tlbcmpx= options) or dirty TLB miss (the tlbdmpn= options). Specify a positive integer for n. The default n depends on your system hardware.

The following arguments control the prefetch operation:

pf1[= (ON|OFF)]

pf2[= (ON|OFF)]

pf3[= (ON|OFF)]

pf4[= (ON|OFF)]

Selectively disables and enables prefetching for cache level x, for pfx[= (ON|OFF)]

When -r10000 or -r12000 are in effect, pf1=ON and pf2=ON by default. At any other -rn setting, OFF is in effect for all cache levels.

prefetch=n

Specifies levels of prefetching. The default is 1 when it is supported and the default is 0 when not supported.

n can be one of the following:

- 0 Disables all prefetching.
- 1 Enables conservative prefetching.
- 2 Enables aggressive prefetching.

`prefetch Ahead=n`

Prefetches the specified number of cache lines ahead of the reference. Specify a positive integer for n. The default is 2.

`prefetch Manual[= (ON|OFF)]`

Specifies whether manual prefetches (through directives) should be respected or ignored.

`prefetch Manual=OFF` ignores manual prefetches. This is the default when `-r8000`, `-r5000`, or `-r4000` is in effect.

`prefetch Manual=ON` respects manual prefetches. This is the default when `-r10000` or `-r12000` is in effect.