The MIPSpro Compilers

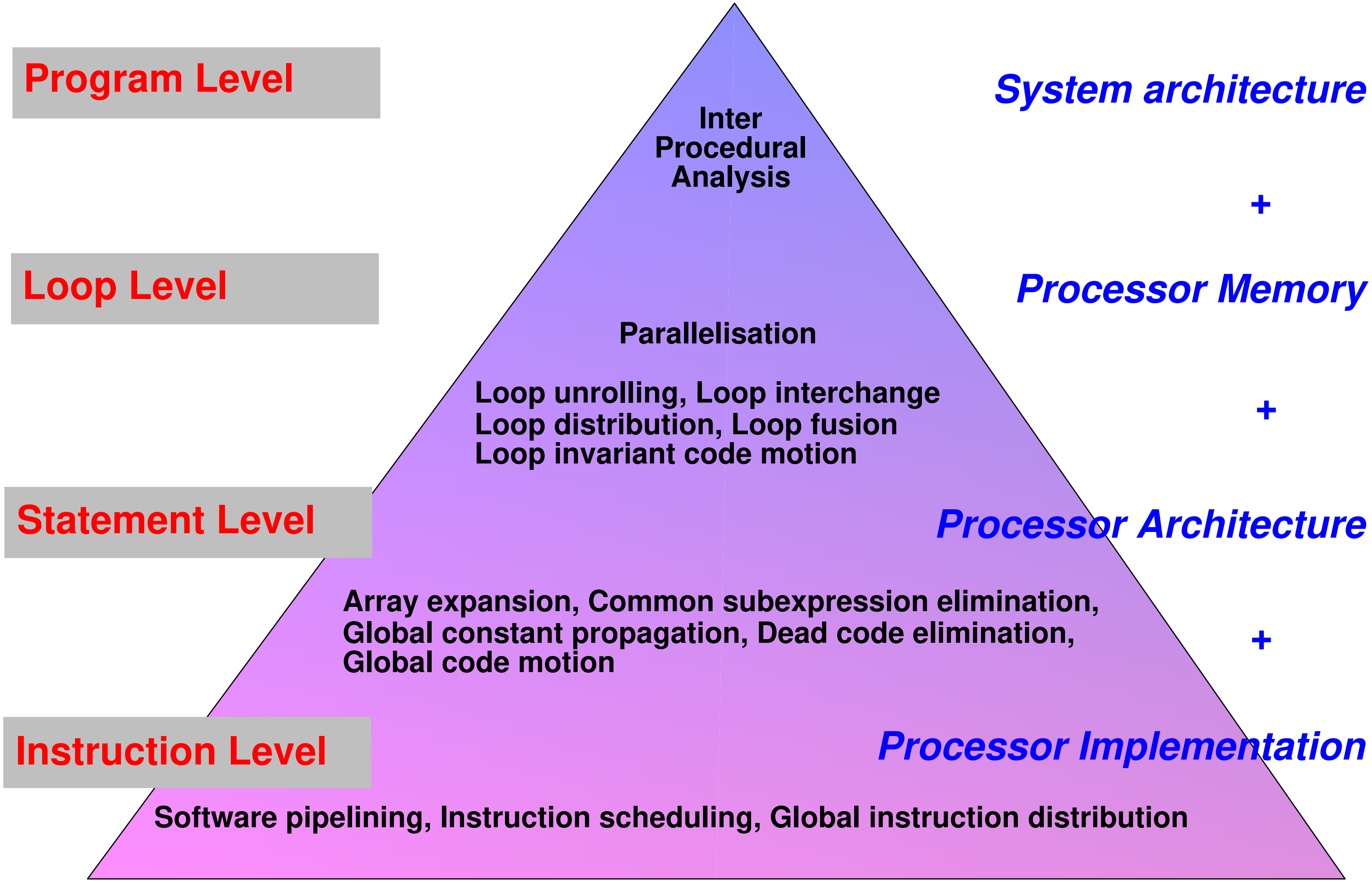
Ruud van der Pas

SiliconGraphics

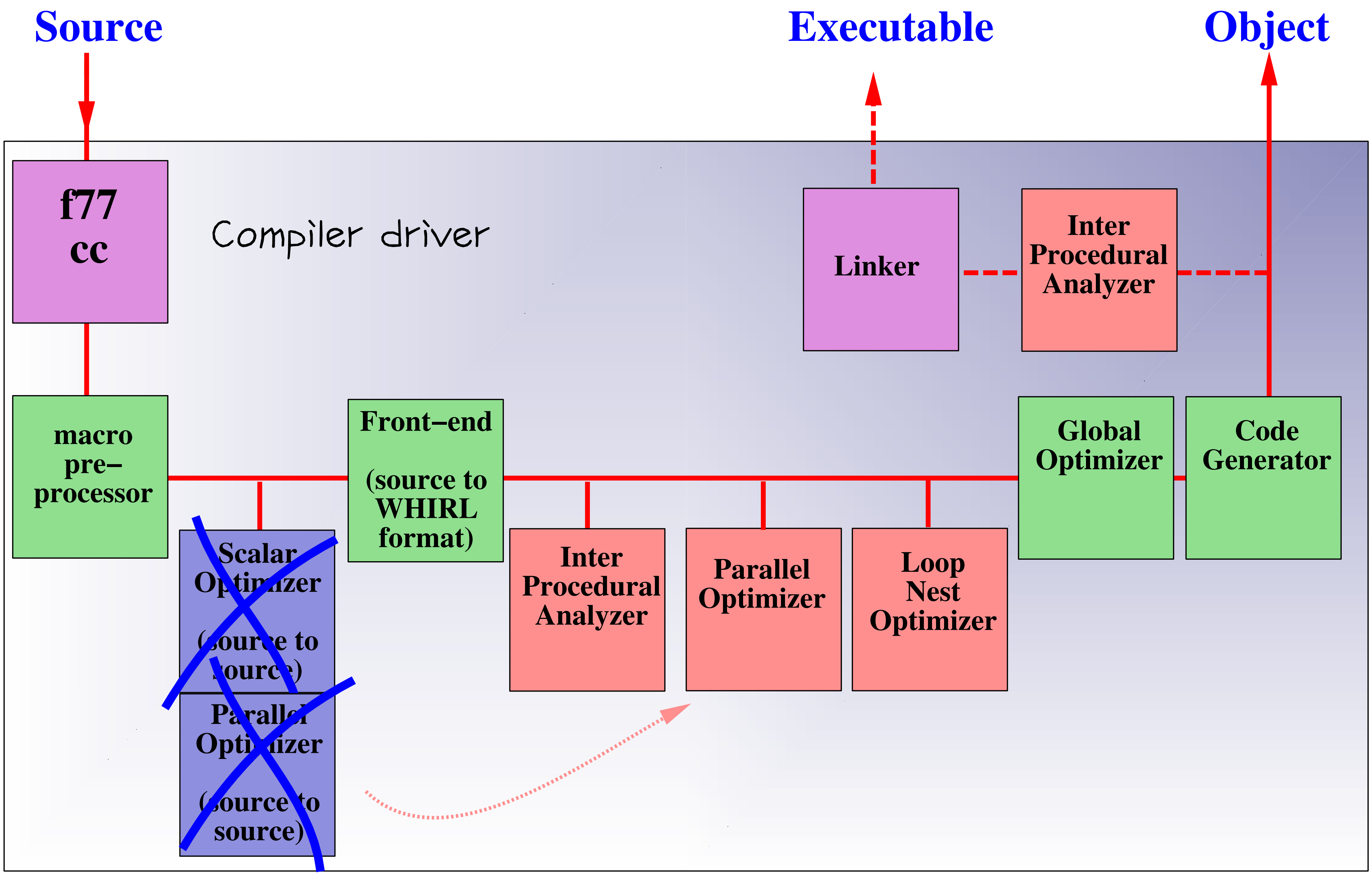
European High Performance Computing Team

Introduction

MIPSpro Compiler Overview



MIPSpro Compiler Components (7.2)



IRIX 6.4 execution/compilation table

	o32 mips I	mips II –32	n32 mips III –n32 –mips3	mips IV –n32 –mips4	n64 mips III –64 –mips3	mips IV –64 –mips4
Indigo, Indigo2(R4K), Indy/Challenge S/M (R4K) ***Execution***	✓	✓	✓			
Development		✓	✓	✓	✓	✓
Indy/Challenge S (R5K) ***Execution***	✓	✓	✓	✓		
Development		✓	✓	✓	✓	✓
Challenge (R4K), Onyx (R4K) ***Execution***	✓	✓	✓		✓	
Development		✓	✓	✓	✓	✓
Pwr Indigo2, PwrChall, PwrOnyx, all R10K platforms ***Execution***	✓	✓	✓	✓	✓	✓
Development		✓	✓	✓	✓	✓

The O2, running IRIX 6.3, supports execution of –n32 and development of –n32/–64

Compiler Optimisations

Global Optimizer

- dead code elimination
- copy propagation
- loop normalization (i.e. stride 1 and one induction variable)
- alias analysis (i.e. memory location referenced many ways)
- strength reduction (i.e. loop counter as array index replaced by pointer arithmetic)
- redundant calculations elimination
- induction variable recognition/simplification/elimination

Compiler Optimisations

Loop Nest Optimizer

- loop unrolling (inner/outer)
- loop interchange
- loop fusion/fission
- loop blocking
- memory prefetch
- software pipelining

Inter Procedural Analyzer

- intra–file inlining (automatically!)
- dead function elimination
- dead variable elimination
- internal padding of common arrays (Fortran only)
- inter procedural constant propagation

Summary of Options

Options – Porting/1

Option	Functionality
–n32	Invoke the MIPSpro compiler, use 32 bit addressing *
–64	Invoke the MIPSpro compiler, use 64 bit addressing
–32	Invoke the 32 bit ucode compiler; 32 bit addressing

There are two more ways that can be used to define the ABI:

- *The environment variable "SGI_ABI" can be set to –n32, –64 or –32*
- *The ABI, ISA and processor type are defined in a file called* ***compiler.defaults***
  One can set environment variable "COMPILER_DEFAULTS_PATHS" to a list of paths where the compiler will look. If this environment variable is not set, or if this file is not found, the compiler will check ***/etc/compiler.defaults***

Example:

```
ruud@itv> cat /etc/compiler.defaults
-DEFAULT:abi=n32:isa=mips4:proc=r10k
```

Options – Porting/2

Option	Functionality
–d8/d16	Double as 8 or 16 bytes
–r8	Convert REAL to REAL*8 and COMPLEX to COMPLEX*16 *)
–i8	Convert INTEGER to INTEGER*8 and LOGICAL to 64 bit *)
–static	Local variables: preserve values and initialize to zero
–col[72|120]	Source line is 72, respectively 120 columns
–mips"n"	Use specified MIPS instruction set (n=1,2,3,4)
–D*name*	Define *name* for the cpp preprocessor
–I*dir*	Define include directory to be *dir*
–align"n"	Force alignment to be on an n=8,16,32,64,128 bit boundary
–G0	Increase tables for static data and program addresses
–xgot	Increase tables for static data and program addresses
–g	Create symbol table for debugging (–g3: optimized code)
–listing	Produce a compiler listing in file with extension .L
–show	Put compiler in verbose mode
–version	Show compiler version

*) *Explicit sizes are preserved, i.e. REAL*4 remains to be 32 bit*

Options – Performance

Option	Functionality
–r10000	Generate optimal instruction schedule for the R10K
–r8000	Generate optimal instruction schedule for the R8K
–r5000	Generate optimal instruction schedule for the R5K
–O[0|1|2|3]	Set optimization level to be [0 | 1 | 2 | 3]
–Ofast	Combines several performance options into one
–mp	Enable the multi–processing directives
–mpio	Fortran: Support I/O from a parallel region
–pfa	Invoke Fortran or C auto–parallelizer
–OPT:*key=val*	Class of additional optimizations
–SWP:*key=val*	Control software pipelining
–TENV:*key=val*	Control target environment
–TARG:*key=val*	Control target architecture
–FLIST/CLIST:*key=val*	F77/C listing control
–LIST:*key=val*	Options to control the listing
–DEBUG:*key=val*	Debugging options
–IPA:*key=val*	Inter Procedural Analyzer control
–INLINE:*key=val*	Inliner control
–LNO:*key=val*	Loop Nest Optimizer control

Optimisation Group Options

Group Options

The following group options are defined:

–OPT:*key=val*	Class of additional optimisations
–TARG:*key=val*	Control target architecture
–TENV:*key=val*	Control target environment
–FLIST/CLIST:*key=val*	F77/C listing control
–LIST:*key=val*	Options to control the listing
–DEBUG:*key=val*	Debugging options
–INLINE:*key=val*	Inliner control
–IPA:*key=val*	Inter Procedural Analyzer control
–LNO:*key=val*	Loop Nest Optimizer control

In general multiple {key=val} combinations are colon separated:

–OPT:*roundoff=2 : alias=restrict*

Note: Exception is –INLINE: keywords are comma separated

Compilers and Performance

Compilers may want to do things which are:

- better for performance, but .....
- possibly altering the numerical behaviour

Examples:

Replacing a division by a fixed variable with a multiplication

Changing the order of a summation

The following two options give the compiler more freedom, and are usually very safe with respect to the numerical results

Recommended Options

–OPT:IEEE_arithmetic=*level*

level=1

Do not perform transformations which may degrade the IEEE floating–point accuracy

level=2

Allow some code transformations which may not conform to IEEE–754 floating–point standard, such as reciprocal and reciprocal square root

level=3

Conformance to IEEE–754 is not required, so allow any valid *mathematical* transformation

Recommended Options

–OPT:roundoff=*level*

level=0

Do not perform transformations which may change the numerical behaviour

level=1

Allow code transformations which have limited effects on the accuracy of the result (in the last 1–2 bits of the mantissa)

level=2

Allow more transformations such as association and distribution

level=3

Allow any mathematically valid transformation, even if it may cause accumulation of roundoff errors

Compilers and Dependencies

Compilers may suspect a possible data dependency, which is not there:

- ***indirectly addressed arrays***
- ***possibly aliased data*** **(equivalence, common blocks, pointers, ...)**

The following option gives the compiler additional information on what sort of dependency exists

This helps to generate faster code

Compilers and Dependencies

–OPT:alias= [any | typed | unnamed | restrict]

any

Default: Any pair of memory references may be aliased

typed

Assume that any pair of memory references that reference distinct data types (like pointer to an integer and a pointer to a float), are not aliased

```
void dbl(int *i,float *f) {
      *i = *i + *i;
      *f = *f + *f;
}
```

unnamed

Assume that pointers never point to unnamed objects

```
float g;
void dbl(float *f) {
       g =  g +  g;
      *f = *f + *f;
}
```

restrict

Assume that different pointers never point to the same memory area

Target Option (TARG) Group

–TARG:platform=ipxx (xx=19,20,....) *

Identify the target platform. Used by the compiler to obtain specific machine characteristics.

–TARG: madd=(ON|OFF)

Default=ON for MIPS IV

Can be used to determine whether rounding differences are caused by the MADD instruction which is more accurate than MUL+ADD.

–TARG: fp_precise=(ON|OFF)

Default=OFF

Meaningful for R8000 only, it forces precise fp mode at run time.

–TARG:processor=(r4000|r5000|r8000|r10000)

Used for instruction scheduling

*) Use "% hinv -c processor" to obtain this information

Target Environment (TENV) Group

–TENV:[large|small]_GOT

Default=small_GOT.

Choose the size of the global offset table (contains addresses of static data and subprograms). The default is a 64 KB table where all data can be accessed by a single load instruction.

Target Environment (TENV) Group

−TENV:X=(0,1,2,3,4,5)

Default=1

Specify level of enabled exceptions:

- **0 Turn off speculation code motions (IEEE−754 exceptions=ON)**
- **1 Perform speculation safe code motions assuming that inexact and underflow IEEE−754 exceptions are off.**
- **2 Disable all IEEE−754 exceptions except divide−by−zero.**
- **3 Disable all IEEE−754 exceptions.**
- **4 Disable (ignore) memory exceptions.**
- **5 Disable (ignore) all exceptions.**

Notes:

- Use this option with great care!
- The value of "X" affects SWP optimizations

Listing options for Fortran

Get compiler's internal program representation back into F77 code

–FLIST:=(ON|OFF)
  Default=OFF
  Enable/disable listing.

–FLIST:ftn_file=filename
  Default=<source–file–name>.w2f.f
  Redirect output to specified file.

–FLIST:ansi_format=(ON|OFF)
  Default=OFF
  ON: Emit Fortran more than 72 characters wide using tabs

–FLIST:linelength=n
  Default=unlimited
  Set maximum linelength to n characters

–FLIST:emit_pfetch (still undocumented)
  Default=OFF
  Show prefetch operations generated

Listing options for C

Get compiler's internal program representation back into C code.

–CLIST:=(ON|OFF)
  Default=OFF
  Enable/disable listing

–CLIST:dotc_file=filename
  Default=<source–file–name>.w2c.c
  Redirect output to specified file

–CLIST:doth_file=filename
  Default=<source–file–name>.w2c.h
  File into which file–scope declarations will be placed

–CLIST:linelength=n
  Default=unlimited
  Set maximum linelength to n characters

–CLIST:emit_pfetch (still undocumented)
  Default=OFF
  Show prefetch operations generated

Emit Prefetch – *Example Output*

```
          DO _i_0 = 1, 50000, 1
C           PREFETCH(268870128) c(_i_0) OFFS=256
C              write strid1=0 strid2=32 conf=3

C           PREFETCH(268869936) c(_i_0) OFFS=64
C              write strid1=8 strid2=0 conf=3

C           PREFETCH(268869744) b(_i_0) OFFS=256
C              write strid1=0 strid2=32 conf=3

C           PREFETCH(268869552) b(_i_0) OFFS=64
C              write strid1=8 strid2=0 conf=3

C           PREFETCH(268869432) a(_i_0) OFFS=256
C              write strid1=0 strid2=32 conf=3

C           PREFETCH(268869360) a(_i_0) OFFS=64
C              write strid1=8 strid2=0 conf=3
C           prefetch (ptr, lrnum): 1st <268869360, 1>2nd <268869432, 1>
            a(_i_0) = 2.0
            b_1 = 2.0
C           prefetch (ptr, lrnum): 1st <268869552, 1>2nd <268869744, 1>
            b(_i_0) = b_1
            a_1 = (b_1 * 2.0)
C           prefetch (ptr, lrnum): 1st <268869360, 1>2nd <268869432, 1>
            a(_i_0) = a_1
C           prefetch (ptr, lrnum): 1st <268869936, 1>2nd <268870128, 1>
            c(_i_0) = ((((a_1 * 5.0E-01) + 1.0)) * a_1) + 1.0)
          END DO
```

reference

+ => address

offset in bytes

read/write field

stride for L1 and L2

"confidence number"

General Listing options

List the compiler options used

–LIST:=(ON|OFF)

Default=OFF

Enable/disable listing which is written to <source–file–name>.l

–LIST:options=(TRUE|FALSE)

Default=FALSE

Get the group (TARG, TENV, ...) options

–LIST:all_options ***(as of release 7.2)***

Get about all the options (many!)

–listing

Gives a source listing (extension .L)

Note: this is not a group option

Listing options – Examples/1

```
% f77 -LIST:=on -O1 -TENV:X=3 -64 -mips3 -TARG:processor=r8000 -c main.f
% cat main.l

-------------------------------------------------------------
 Compiling main.f (/tmp/ctmB.BAAa003pd)
-------------------------------------------------------------


-------------------------------------------------------------
 Options:
-------------------------------------------------------------
  Target:R8000, ISA:mips3, Pointer Size:64
  -O1    (Optimization level)
  -g0    (Debug level)
  -m1    (Report warnings)
-------------------------------------------------------------
```

Listing options – Examples/2

```
% f77 -LIST:=on:options=true -O1 -TENV:X=3 -64 -mips3 -TARG:processor=r8000 -c main.f
% cat main.l

------------------------------------------------------------
 Compiling main.f (/tmp/ctmB.BAAa0041z)
------------------------------------------------------------
------------------------------------------------------------
 Options:
------------------------------------------------------------
  Target:R8000, ISA:mips3, Pointer Size:64
  -O1    (Optimization level)
  -g0    (Debug level)
  -m1    (Report warnings)
------------------------------------------------------------
 Group options are marked with '#' if changed,
 or with '=' if explicitly set to default value
------------------------------------------------------------
 -LANG Option Group
------------------------------------------------------------
#                      = f77
------------------------------------------------------------
------------------------------------------------------------
 -LIST Option Group
------------------------------------------------------------
#                      = ON
# options              = ON
------------------------------------------------------------
------------------------------------------------------------
 -TARG Option Group
------------------------------------------------------------
# abi                  = 64
# processor            = r8000
# t5_ll_sc_bug         = ON
------------------------------------------------------------
------------------------------------------------------------
 -TENV Option Group
------------------------------------------------------------
# X                    = 3
# io_library           = mips
------------------------------------------------------------
```

Listing options – Examples/3

```
% f77 -LIST:all_options source.f
```

*Gives all (internal) switches set in the following option groups: **

–DEBUG	Assist debugging
–INLINE	Control subprogram inlining
–IPA	Control interprocedural analysis and optimization
–LANG	Control source language interpretation
–LIST	Control the listing file
–OPT	Control general optimization
–LNO	Control loop nest optimization
–TARG	Specify the target machine characteristics
–TENV	Set or assert target environment characteristics
–FLIST	Control listing of transformed f77 source
–CLIST	Control listing of transformed C source
–PURPLE	Control program region extraction process
–PROMP	Control listing mp transformations

*) Note: these are not always documented

Listing options – Examples/4

```
      PROGRAM ERRORS

      IMPLICIT NONE

      INTEGER  N
      PARAMETER (N=1000)

      INTEGER  I, J
      REAL     A(N,N)

C$DOACROSS LOCAL(J,I) SHARED(A)
      DO J = 1, N
         DO I = J, N
            A(I,J) = I+J
         END DO
      END DO

      PRINT *,This is a typo',A(N,N)

      DO 1000 I = 1, N
         A(I,I) = 1.0
      END DO

      A(N+1,N) = 2.0

      STOP
      END
```

- This program contains several syntax errors
- We compile it with the following options:

–listing	get a listing
–mp	for parallelisation
–fullwarn	maximum diags.

- Note that we rely on the ABI, ISA and processor settings in /etc/compiler.defaults !

Listing options – Examples/4

```
% f77 -listing -mp -fullwarn errors.f
```

```
"errors.f", line 18: error(2106): identifier "Thisisatypo" has not been
            explicitly typed

        PRINT *,This is a typo',A(N,N)
                ^
"errors.f", line 18: error(2022): missing closing quote

        PRINT *,This is a typo',A(N,N)
                              ^
"errors.f", line 18: error(2045): extra text after expected end of
            statement

        PRINT *,This is a typo',A(N,N)
                              ^
"errors.f", line 22: error(2228): END DO label does not match label on
the corresponding DO

        END DO
        ^
"errors.f", line 24: warning(2134): subscript out of range

        A(N+1,N) = 2.0
          ^
"errors.f", line 20: error(2161): label "1000" was referenced but not
            defined
        DO 1000 I = 1, N
           ^
5 errors detected in the compilation of "errors.f".
```

Listing options – Examples/4

The errors.L file:

/etc/compiler.defaults

```
Compiling errors.f.

Options:
  Target:R10000, ISA:mips4, Pointer Size:32
  -O0    (Optimization level)
  -g0    (Debug level)
```

```
L 1 "errors.f"
N       PROGRAM ERRORS
S
N       IMPLICIT NONE
S
N       INTEGER  N
N       PARAMETER (N=1000)
S
N       INTEGER  I, J
N       REAL     A(N,N)
S
NC$DOACROSS LOCAL(J,I) SHARED(A)
N       DO J = 1, N
N          DO I = J, N
N             A(I,J) = I+J
N          END DO
N       END DO
```

```
S
N       PRINT *,This is a typo',A(N,N)
E "errors.f" 18 15 S
N       DO 1000 I = 1, N
E "errors.f" 18 29 E "errors.f" 18 29 N
A(I,I) = 1.0
N       END DO
S
N       A(N+1,N) = 2.0
E "errors.f" 22 7 W "errors.f" 24 9 S
N       STOP
N       END
E "errors.f" 20 10 S
```

–DEBUG Option Group – Runtime/1

–DEBUG:div_check=(0,1,2,3)

Default=1
Insert checks for integer divide by 0 and overflow.
0: no check
1: divison by 0 check
2: overflow check
3: division by 0 and overflow check

–DEBUG:subscript_check=(ON | OFF)

Default=OFF
Runtime check for subscripts out of range

–DEBUG:verbose_runtime=(ON | OFF)

Default=OFF
For runtime checks report source routine and line number instead of only trapping.
Limited support for C/C++ for pre–IRIX 6.5 releases. See also the DEBUG_group (5) man page

–DEBUG Option Group – Runtime/2

–DEBUG:trap_uninitialized=(ON | OFF)

Default=OFF

Force all un–initialized stack, automatic and dynamically allocated variables to be initialized with 0xFFFA5A5A.

When this value is used as a floating point variable, it is treated as a floating point NaN and it will cause a floating point trap.

When it is used as a pointer, an address or segmentation violation will most likely occur.

–DEBUG:varargs_interface_check=(ON | OFF)

Default=ON

Emit subprogram interface descriptors for all extern varargs routines (C and C++) and calls to non–prototyped routines with FP actual parameters, and check the calls in the linker and rld for consistency.

–DEBUG:varargs_prototypes=(ON | OFF)

Default=ON

Surpress failures caused by missing prototypes for vararg routines that have FP arguments. Will cost some performance.

–DEBUG Option Group – Compiletime

–DEBUG:fullwarn=(ON | OFF)
Default=ON
Emit all informational warnings, including remarks.

–DEBUG:error=n1 [,n2, ...]
Treat warning/remarks with message id n1, n2, .. as errors.

–DEBUG:remark=n1 [,n2, ...]
Treat the problems with message id n1, n2, .. as remarks.
This means they are not printed, unless fullwarn is also set.

–DEBUG:surpress=n1 [,n2, ...]
Surpress the messages with id n1, n2, ..

–DEBUG:warning=n1 [,n2, ...]
Treat the problems with messages id n1, n2, .. as warnings

–DEBUG:woff=n1 [,n2, ...]
Turn of the messages with id n1, n2, ..

Compiler Based Inlining

Benefits

- Exposes larger context to loptimisation phases
- Eliminates call overhead
- Eliminates aliases and resolves otherwise unknown side effects

Disadvantages

- Compilation time increases
- Creates harder register allocation problem
- Increases program text size

Restrictions

- No varargs routines
- No recursive routines
- No mismatched parameter types
- No static (save) local variables

There are two types of Inlining

Manual inliner; use –INLINE

- Only inlines routines within the same source file
- Performs no automatic inlining: you must tell it what to do!
- Inlines according to inline keyword and –INLINE options

IPA's automatic inliner; use –IPA

- Will inline routines across file boundaries
- Automatically picks which calls to inline
- Prefers the following:
  - small callees
  - small callers
  - calls in innermost loops
  - leaf routines
  - calls executed frequently (with –feedback)
- Stops inlining if –Olimit exceeded or size of program doubles

The Standalone Inliner – Options/1

–INLINE
: Enable inlining with default option settings (and heuristic)

–INLINE:all
: Attempt to inline all routines (which are not excluded) within the source file

–INLINE:none
: Change default inlining heuristic and attempt to inline only routines specified in a must clause

–INLINE:must=name1[,name2...]
: Always attempt to inline the named routines

–INLINE:never=name1[,name2...]
: Never attempt to inline the named routines

The Standalone Inliner – Options/2

–INLINE:list=(ON|OFF)

Default=OFF

Verbose mode: enable/disable listing of inlining actions

–INLINE:dfe=(ON|OFF)

Default=OFF

Enable/disable dead function elimination

–INLINE:specfile=filename

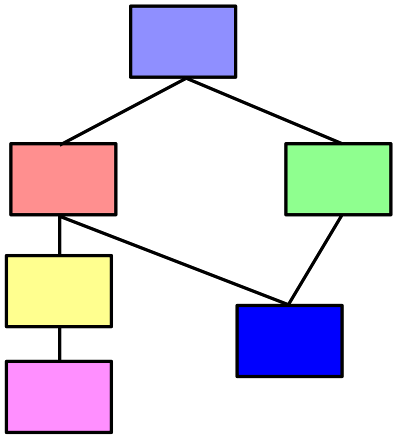
Read additional inlining options in "filename"

Inter–Procedural Analysis (IPA)

- *Analyze and optimise code dependencies between separate program modules (subroutines, functions, ...):*

This will allow for a much broader data analysis by the compiler

As a result, it is able to do much more optimisations



- *IPA operates by postponing much of the compilation process until link time when all modules can be analysed together*

- *The ".o" files now contain an intermediate representation code (WHIRL) instead of relocatable code*

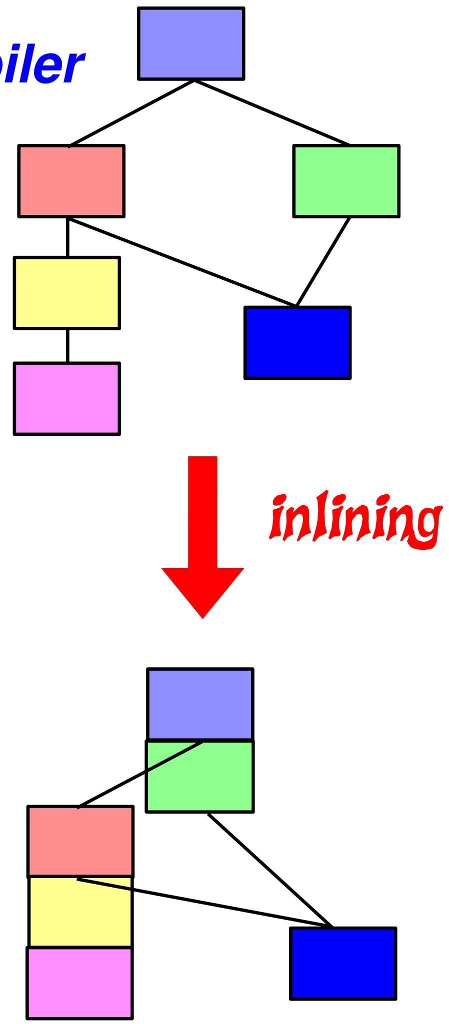
IPA optimisations

- **Automatic inlining**
  Calls to a procedure are replaced by a suitably modified copy of the called procedure's body inline, even if the callee is in a different source file
- **Comon block array padding**
  Global arrays in Fortran may be padded, i.e. increased in size in order to reduce cache conflicts
- **Constant propagation**
  Formal parameters which always have a particular constant value can be replaced by the constant, allowing additional optimization.
- **Dead variable and function elimination**
  Variables that are never used and functions which are never called can be removed from the program, improving memory utilization.
- **Global name optimizations**
  Global names in shared code must normally be referenced via addresses in a global table, in case they are defined or preempted by another DSO . If the compiler knows it is compiling a main program and that the name is defined in another of the source files comprising the main program, an absolute reference can be substituted, eliminating a memory reference.

  Similarly, IPA can optimize the data items which can be referenced by simple offsets from the GP register instead of depending on the user to provide an ideal value of the –G option

Automatic Inlining

- *Amongst one of the most powerful features of IPA is the auto–inlining capability*
- *A semi–standalone inliner is part of the compiler*
- *When used in conjuction with IPA, this inliner performs at it's best*
- *For inter–module inlining one needs IPA*
- *However, for intra inlining (i.e. modules within the same source file) one can use the inliner without activating IPA*
- *Examples will be given shortly*

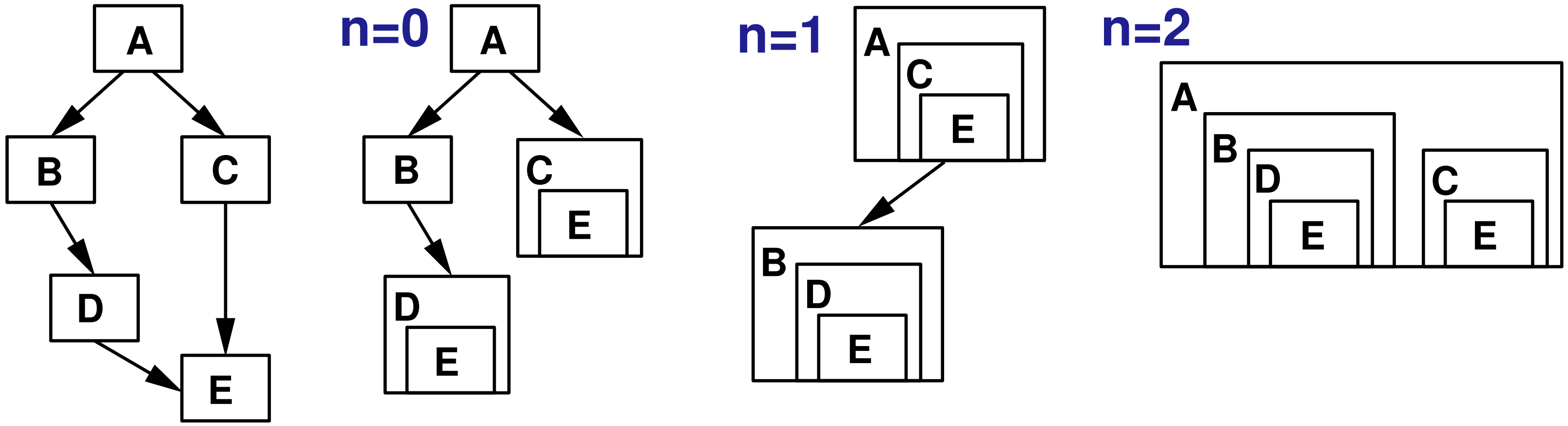


Automatic inlining with IPA

Stops inlining if –Olimit exceeded or size of program doubles

Controlled with the following options:

- **–INLINE:all, –INLINE:none, –INLINE:must, –INLINE:never**
- **–IPA:space=<n>** where n is the largest acceptable % increase in code size from inlining
- **–IPA:forcedepth=n** inline all routines at depth <= n in the call graph *regardless of space limit*; leaf routines @ depth 0
- **–IPA:inline=off** turns off inlining



IPA options – 1

–IPA

Invoke IPA with all default settings

–IPA: <inlining options>

Refer to the standalone INLINER options

–IPA:inline=(ON|OFF)

Default=ON

Perform inter–file subprogram inlining during main IPA processing

Does not affect the standalone inliner

–IPA:plimit=n

Default=2500

Stop inlining into a particular subprogram once it reaches size n in the intermediate representation

–IPA:space=n

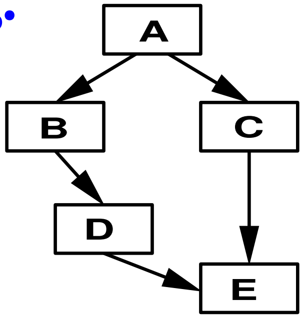
Default=100%

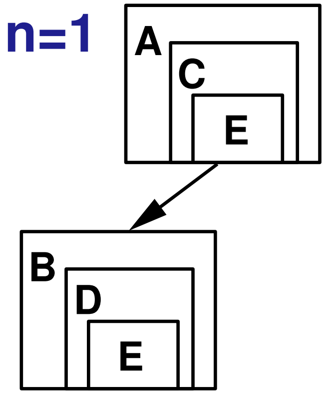
Sets code size overheads limits (stop IPA after code has increased with n%)

IPA options – 2

–IPA:forcedepth=n

Instead of the default inlining heuristics, attempt to inline all functions at a depth of at most n in the callgraph, where functions which make no calls are at depth 0, those which call only depth 0 functions are at depth 1, and so on. Ignore the default heuristic limits on inlining. See also maxdepth.





–IPA:maxdepth=n

In addition to the default inlining heuristics, don't attempt to inline functions at a depth of more than n in the callgraph, where functions which make no calls are at depth 0, those which call only depth 0 functions are at depth 1, and so on. Inlining remains subject to overriding limits on code expansion. See also forcedepth, space and plimit.

IPA options – 3

–IPA:max_job=n

Default=4 (MP system) and 2 on a uniprocessor

Limit the maximum parallelism when invoking the compiler back end after IPA to at most n compilations running at once

–IPA:echo=(ON|OFF)

Default=OFF

Enable/disable echoing of compile and link commands issued

User Land

Compiler Land

–IPA:dfe=(ON|OFF)

Default=ON

Enable/disable dead function elimination, i.e. removal of subprograms which are never called from the program

–IPA:dve=(ON|OFF)

Default=ON

Enable/disable dead variable elimination, i.e. removal of variables which are never referenced from the program

IPA options – 4

–IPA:alias=(ON|OFF)

Default=OFF

Enable/disable alias/mod/ref analysis. See also –IPA:opt_alias

–IPA:opt_alias=(ON|OFF)

Default=OFF

Transfer the results of alias/mod/ref analysis to the global optimizer. Forces –IPA:alias=ON if set.

–IPA:addressing=(ON|OFF)

Default=OFF

Enable/disable the analysis of address operator usage. –IPA:alias=ON is a prerequisite

–IPA:aggr_cprop=(ON|OFF)

Default=OFF

Enable/disable aggressive constant propagation analysis to avoid passing constant parameters

IPA options – 5

–IPA:cprop=(ON|OFF)

Default=ON

Enable/disable interprocedural constant propagation, i.e. identify formal parameters which always have a specific constant value.

See also –IPA:aggr_cprop

–IPA:cgi=(ON|OFF)

Default=ON

Enable/disable constant global variable identification, i.e. mark non–scalar global variables which are never modified as constant, and propagate their constant values to all files

–IPA:picopt=(ON|OFF)

Default=ON

Perform PIC optimizations, e.g. identify names which cannot be preempted

IPA options – 6

–IPA:autognum=(ON|OFF)

Default=ON

Determine the optimal value of the –Gnum option, i.e. identify a size bound below which data can be allocated relative to the global pointer and accessed cheaply. This optimization is turned off when –multigot is specified in the linker command line. See also –IPA:Gnum

–IPA:Gnum=n

Default=unlimited

User specified G num

–IPA:Gspace=n

Default=64K (maximum)

User specified size (in bytes) for the area where IPA can allocate data that can be referenced relative to the global pointer.

Example – IPA and Inlining

Structure of the example code:

```
main
{
      loop over
            call do_work

      call another_one
}
```

We want to inline this one

...., but NOT this one

Two source files:

Fortran	C
main.f	main.c
subs.f	funcs.c (contains do_work and another_one)

Example – IPA and Inlining

File main.f:

```
program main
implicit none
integer N
parameter (N=50000)
integer i, void, another_one
real     a(N), b(N), c(N)

i = 10

do i = 1, N
   a(i) = 2.0
   b(i) = 2.0
end do

do i = 1, N
   a(i) = b(i) * 2.0
   call do_work(a(i),c(i))
end do

i = 1957
void = another_one(i)

do i = 1, N
   c(i) = c(i) + b(i)*2.0
end do

print *,'c(',1,')=',c(1),' c(',N,')=',c(N)

stop
end
```

File subs.f:

```
subroutine do_work(x,y)
implicit none
real x, y

y = 1.0 + x * (1.0 + x * 0.5)

return
end

integer function another_one(i)
implicit none
integer i
print *,'i=',i
return
end
```

Example – IPA and Inlining

```
% f77 -c -64 -mips4  -O2
  -INLINE:LIST=on:must=do_work:never=another_one
  -FLIST:=on:ftn_file=main.list
  -LIST:=on:options=TRUE -IPA main.f

% f77 -c -64 -mips4  -O2
  -INLINE:LIST=on:must=do_work:never=another_one
  -FLIST:=on:ftn_file=subs.list
  -LIST:=on:options=TRUE -IPA subs.f

% f77 -o inline -64 -mips4
  -INLINE:LIST=on:must=do_work:never=another_one
  -IPA main.o subs.o

another_one_ not inlined into MAIN__: user requested not
to inline

do_work_ inlined into MAIN__
```

Example – IPA and Inlining

File main.list

```
        PROGRAM MAIN
        IMPLICIT NONE
C
C       **** Variables and functions ****
C
        REAL*4 a(50000_8)
        REAL*4 b(50000_8)
        REAL*4 c(50000_8)
        INTEGER*4 i
        EXTERNAL another_one
        INTEGER*4 another_one
C
C       **** Temporary variables ****
C
        INTEGER*4 tmp0
C
C       **** statements ****
C
        DO i = 1, 50000, 1
          a(i) = 2.0
          b(i) = 2.0
        END DO
        DO i = 1, 50000, 1
          a(i) = (b(i) * 2.0)
8         CONTINUE
          c(i) = ((((a(i) * 5.0E-01) + 1.0)) * a(i)) + 1.0)
7         CONTINUE
        END DO
        i = 1957
        tmp0 = another_one(i)
        DO i = 1, 50000, 1
          c(i) = (c(i) +(b(i) * 2.0))
        END DO
        PRINT *, 'c(', 1, ')=', c(2 - 1), ' c(', 50000, ')=', c(50001 - 1)
        STOP
        END ! MAIN
```

Example – Performance

```
% f77 -o no_inline -O3 -64 -mips4 main.f subs.f
% time ./no_inline
 i=          1957
 c(            1)=   17.00000   c(      5000000)=   17.00000
0.971u 0.421s 0:01.43 97.2% 0+0k 0+1io 0pf+0w

% f77 -o inline -O3 -64 -mips4 -IPA \
      -INLINE:never=another_one:list=on main.f subs.f
main.f:
subs.f:
do_work_ inlined into MAIN__: forced because of small
size (12)
another_one_ not inlined into MAIN__: user requested not
to inline
do_work_ inlined into MAIN__

% time ./inline
 i=          1957
 c(            1)=   17.00000 c(      5000000)=   17.00000
0.394u 0.271s 0:00.68 97.0% 0+0k 0+1io 0pf+0w
```

Almost 3 times faster !

About IPA

- IPA without inlining should always benefit run–time performance
- IPA with inlining can help a lot, or hurt run–time performance

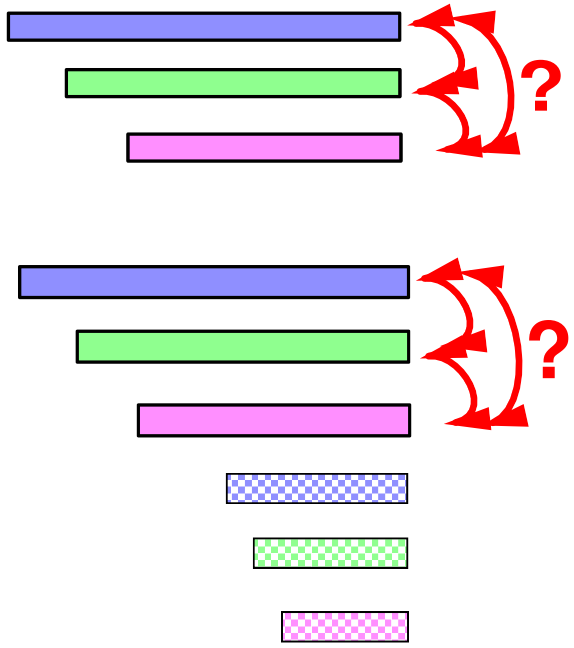
Feedback files may be used with IPA to get better inlining results

The Loop Nest Optimizer (LNO)

- Goal is *to make things simple* for most users yet flexible for power users
- Run by default at –O3 or –Ofast
- Turned off using –LNO:opt=0
- Almost every optimisation individually controlled using options and pragmas
- Pragmas disabled with –LNO:ignore_pragmas

The Loop Nest Optimizer (LNO)

- *LNO performs many loop based optimisations*



Loop unrolling

Loop interchange

Loop fusion

Loop fission

Loop blocking for the cache

Prefetching

LNO: *How to Control It – 1*

- LNO performs many optimisations and transformations
- On a selective basis the different phases can be switched on/off, or sometimes controlled in a more subtle manner
- Example:

  –LNO:blocking=OFF to switch of loop blocking

- The specific settings are dependent on:
  - Optimisation level (i.e. –O3)
  - Target architecture (e.g. prefetching on R10K only)

LNO: *How to Control It – 2*

- LNO can be manually controlled with compiler options and/or compiler directives (Fortran) or pragmas (C)
- Two control LNO on a global level, use the following option:

```
–LNO:opt=(0,1)
    Default=1
    0: perform minimal optimisations
    1: perform full optimisations
```

- Options are the same on both compilers
- Directives/pragma syntax:

```
Fortran    C*$*      keyword [=value(s)]
C          #pragma   keyword[=value(s)]
```

- The *LNO defaults are chosen very well*, but in certain cases require adaption. How to do this will be the next topic.

LNO: *Data Dependencies*

CDIR$ IVDEP

Ignore dependences in the following inner loop

Comments:

- **Note the difference between the CDIR$ and C*$* syntax!**
- **The interpretation of "dependency" depends upon the following compiler settings:**
  - **Default: given 2 memory references, with at least one loop variant, ignore loop–carried dependences between the two**
  - **With –OPT:cray_ivdep=TRUE, use the Cray semantics. Break all lexically backwards dependencies**
  - **With –OPT:liberal_ivdep=TRUE, break *ALL* dependences**

LNO: *Controlling Loop Blocking*

–LNO:blocking=(ON|OFF)

Default=ON

Enable/disable loop blocking.

–LNO:blocking_size=[n1][,n2]

Specify blocking sizes to use.

C*$* BLOCKING SIZE (n1,n2) or (n1) or (,n2)

Blocking size for L1 (and L2) cache. Applies to the loop immediately following the directive

C*$* NO BLOCKING

Prevent loop blocking optimizations

C*$* BLOCKABLE (I,J,K)

Inform the compiler that the following nested loops can be blocked even if the compiler thinks the transformation would be illegal

LNO: *Manual Blocking – Example*

```
      subroutine amat(x,y,z,n,m,mm)
      real*8 x(1000,1000), y(1000,1000), z(1000,1000)

C*$* BLOCKING SIZE (0,0)
      do k = 1, n
C*$* BLOCKING SIZE (0,200)
         do j = 1, m
C*$* BLOCKING SIZE (0,200)
            do i = 1, mm
               z(i,k) = z(i,k) + x(i,j)*y(j,k)
            end do
         end do
      end do
     end
```

```
      subroutine amat(x,y,z,n,m,mm)
      real*8 x(1000,1000), y(1000,1000), z(1000,1000)
      do jj = 1, m, 200
         do ii = 1, mm, 200
            do k = 1, n
               do j = jj, MIN(m, jj+199)
                  do i = ii, MIN(mm, ii+199)
                     z(i,k) = z(i,k) + x(i,j)*y(j,k)
                  end do
               end do
            end do
         end do
      end do
     end
```

LNO: *Model the Memory Hierarchy!*

LNO can be used to specify the following:

–LNO:cache_size"L"	= x (bytes)
–LNO:line_size"L	= n (bytes)
–LNO:associativity"L"	= n
–LNO:clean_miss_penalty"L"	= n (cycles)
–LNO:dirty_miss_penalty"L"	= n (cycles)
–LNO:is_memory_level"L"	= on | off
–LNO:tlb_dirty_miss_penalty"L"	= n (cycles)

L = 1, 2, 3 or 4

A value of 0 implies there is no cache at this level

LNO: *Loop Interchange*

–LNO:interchange=(ON|OFF)

Default=ON

Enable/disable loop interchange

C*$*INTERCHANGE (I,J,K)

Loops I,J,K (in any order) following this directive are interchanged for I (outer loop), J (middle loop) and K (inner loop). The loops must be perfectly nested or LNO may distribute them to make them so.

The compiler may choose to ignore this directive.

C*$* NO INTERCHANGE

Prevents the compiler from involving the loop directly following this directive in a loop permutation, *or any loop nested within this loop.*

LNO: *Loop Unrolling*

–LNO:outer_unroll="n"

Every outer loop should be unrolled by "n"

This option is mutually exclusive to the following 2 options:

–LNO:outer_unroll_max="n"

Every outer loop should be unrolled by *at most* "n"

–LNO:outer_unroll_prod_max="n"

Default=16

The product of the unroll factor of the loops in a nest should not exceed "n"

–LNO:outer_unroll_further="n" and –LNO:outer_unroll_deep="n"

Controls the unrolling of wind–down loops created because of .... unrolling

C*$* UNROLL ("n")

Unroll the following loop "n" times (inner or outer)

n=0: default unrolling

n=1: no unrolling

LNO: *Loop Interchange and Unrolling*

In general, LNO considers:

- ***Interchange***
- ***Unrolling***
- ***Blocking***

together to achieve best overall performance

LNO: *Loop Fission and Fusion*

–LNO:fission=(0,1,2)

Default=1

0: no fission

1: normal fission

2: fission tried before fusion; attempts to fission inner loops as much as possible

Fusion is preferred if both fission and fusion are set to 1 or 2

–LNO:fusion=(0,1,2)

Default=1

0: no fusion

1: normal fusion

2: fuse outer loops, also partial fusion; try fusion before fission

–LNO:outer[=(ON|OFF)]

Default=ON

Turn outer loop fusion on/off

–LNO:fusion_peeling_limit[=n]

Default=5

Sets the limit for number of loop iterations allowed to be peeled in fusion

LNO: *Directives for Fission/Fusion*

C*$ AGGRESSIVE INNER LOOP FISSION

Fission the following *inner* loop to as many loops as possible

C*$* FISSION [(n)]
C*$* FISSIONABLE

Fission the enclosing "n" level of loops; perform legality test unless there is a FISSIONABLE directive.

C*$* FUSE [(n)]
C*$* FUSEABLE

Fuse the following "n" loops (default: n=2); apply peeling up to specified limit (default=5, see compiler options); perform legality test unless there is a FUSEABLE directive.

C*$* NO FISSION

Do not perform fission on the following loop (inner loops may still be fissioned).

C*$* NO FUSION

Do not fuse the following loop with other loops.

LNO: *Prefetching*

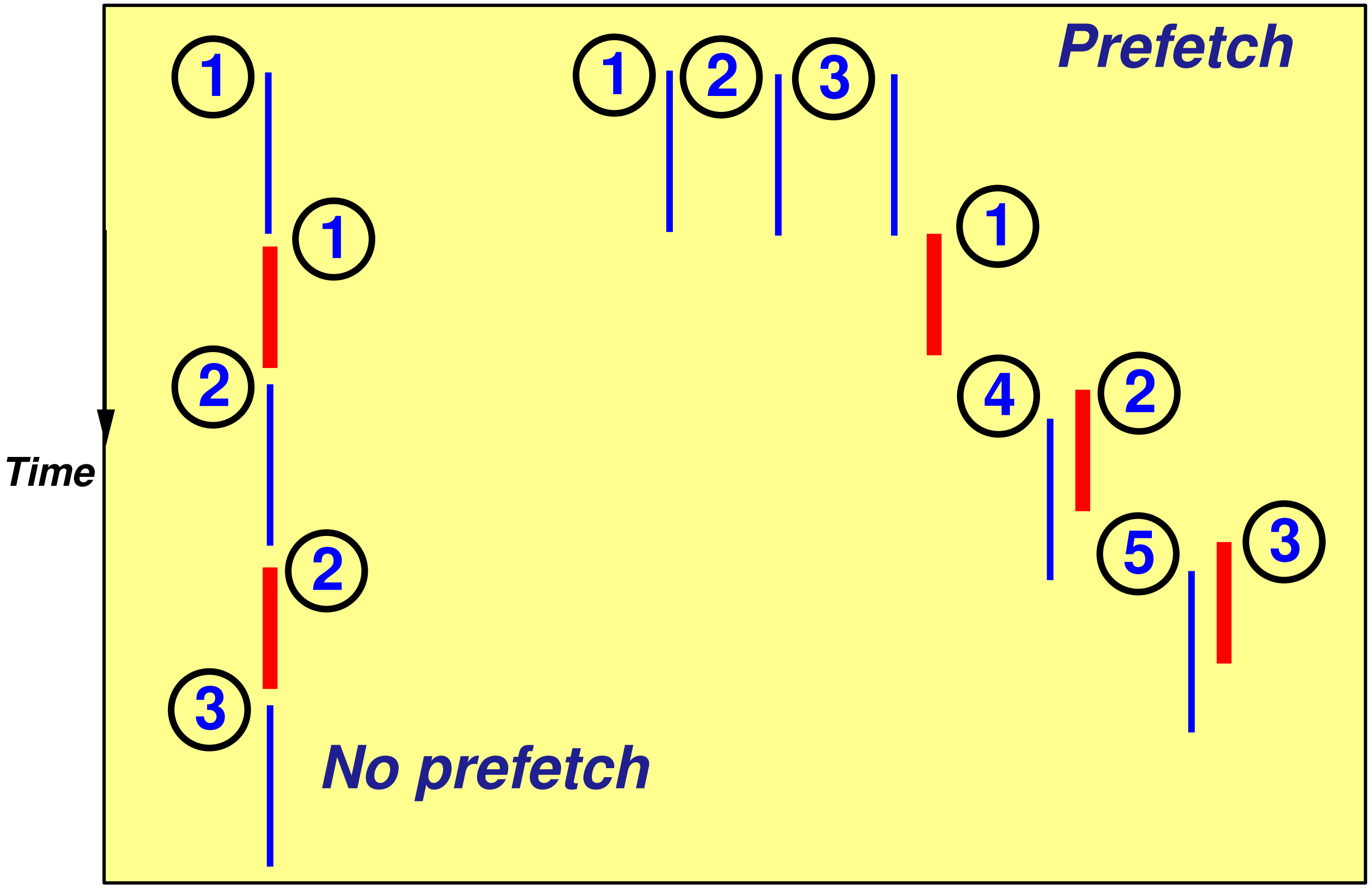
- *One of the most advanced features of the R10K and Origin is the support for non–blocking loads (up to 4)*
- *The MIPS 4 prefetching instruction ("pref") is used to exploit this feature*
- *LNO generates these instructions at –O3*
- *In many cases, LNO does the right thing, but ...*
- *In addition to this (or instead of), the user has full control over prefetching as well*
- *This is called **manual prefetching***

LNO: *Manual Prefetching – Example*

```
c*$* prefetch_ref=a(1)
c*$* prefetch_ref=a(1+16)
          do i = 1, n
c*$* prefetch_ref=a(i+32), stride=16, kind=rd
             sum = sum + a(i)
          end do
```

Cache lines

1	1 ...16
2	17 ... 32
3	33 ...48
4	49 ...54
	...



Pseudo Prefetching

Implicit/Psuedo Prefetch (L2–>L1):

```
for (i=0; i<n; i+=4) {
    a += b[i+0];
    a += b[i+1];
    a += b[i+2];
    a += b[i+3];
}
```

```
for (i=0; i<n; i+=4) {
    t  = b[i+3];
    a += b[i+0];
    a += b[i+1];
    a += b[i+2];
    a += t;
}
```

No instruction overhead, but increased register pressure

This is worse than a possible redundant prefetch:

```
for (i=0; i<n; i++) {
      if ((i % 16) == 0) prefetch b[i+16];
      a = a + b[i];
}
```

LNO: *Prefetch – Options Overview*

–LNO:prefetch="n"

Control the level of prefetching by choosing n:

0	disable prefetch
1	conservative prefetching
2	aggressive prefetching

Default: n=2 for R10000 and n=0 for all other processors

–LNO:prefetch_ahead="n"

Prefetch "n" cache lines ahead of the reference

Default: n=2

–LNO:prefetch_manual[=(ON|OFF)]

Respect (or ignore) manual prefetching

Default: ON for R10000 and OFF for all other processors

–LNO:prefetch_level"n"[=(ON|OFF)]

Selectively enable/disable prefetch for cache level "n", with n=1,2,3 or 4

LNO: *Manual Prefetch – Directives/1*

C*$* PREFETCH ("n1" [,"n2"])

Set prefetch level for cache levels n1 and n2

0: prefetching off (default for all but the R10000)

1: conservative prefetching

2: aggressive prefetching (default when prefetching enabled)

Scope: entire function containing the directive

C*$* PREFETCH_MANUAL ("n")

Use "n" to specify whether manual prefetch is ignored or not

0: ignore manual prefetch (default for MIPS 3 and earlier)

1: respect manual prefetch (default for MIPS 4)

2: aggressive prefetching (default when prefetching enabled)

Scope: entire function containing the directive

C*$* PREFETCH_REF_DISABLE="A" [, size=size_of_array_in_KByte]

Disable prefetching for all references to array "A"

Scope: entire function containing the directive

LNO: *Manual Prefetch – Directives/2*

```
C*$*PREFETCH_REF=array–ref,
C*$*&                          [stride=[str][,str]],
C*$*&                          [level=[lev][,lev]],
C*$*&                          [kind=[rd|wr]],
C*$*&                          [size=[n]]
```

Generates a prefetching instruction

- array–ref = reference to prefetch (ex: A(i,j))
- stride = OPTIONAL
  prefetch every "stride" iterations for this loop (default=1)
- level = OPTIONAL (default=2)
  memory level to prefetch from
  - 1: prefetch from L2 to L1
  - 2: prefetch from memory to L1
- kind = OPTIONAL (default=wr)
- size = OPTIONAL
  size of array referenced (must be a constant in Kbytes)

Scope: no scope, just generate a prefetch instruction on "array_ref"

LNO: *Array Padding*

LNO automatically pads local data

At –O3 or –Ofast, LNO pads common blocks

All routines containing common must be compiled with –O3

Padding is safe as long as the Fortran standard is not violated

```
subroutine sub
common a(512,512), b(512*512)

do i = 1, 2*512*512
   a(i) = 0.0
end
```

Fix violation, use lower optimization level than –O3, or turn off common block padding:

–OPT:reorg_common=off

LNO: *Vector Intrinsics*

```
subroutine vfred(a)
real*8 a(200)
do i = 1, 200
   a(i) = a(i) + cos(a(i))
enddo
end
```

```
             CALL vcos$(a(1), deref_se1_F8(1),
            &      %val((201 - 1)), %val(1),
            &      %val(1))
             DO i = 1, 200, 1
               a(i) = (a(i) + deref_se1_F8(i))
             END DO
```

Almost always faster, but different precision rules

–LNO:vintr=off   Disables vector intrinsics

(currently this ability is limited in C and C++)

Specialized Options

A very special option

–Ofast[=ipxx] xx=*25* (PChall R10K), 27 (O2000)

Default=ip25

A *collection of options* to maximize performance:

> "The selected optimizations may differ between releases of the compiler, and among the supported platforms. They will always enable the full instruction set of the target platform (e.g. -mips4 for an R10000). Although the optimizations are generally safe, they may affect floating point accuracy due to rearrangement of computations."

Currently:

```
-O3, -IPA, -TARG:platform=ipxx, -n32,
-OPT:Olimit=0:roundoff=3:div_split=ON:alias=typed
```

Specific Optimisation Options – 1

–OPT:fast_bit_intrinsics=(ON|OFF)
    Default=OFF
    Enable/disable check for bit count validity for F77 bit intrinsics.

–OPT:fast_sqrt=(ON|OFF)
    Default=OFF
    Enable/disable substitution of "sqrt(x)" by "x*rsqrt(x)" (WARNING sqrt(0.0)=>NaN).

–OPT:IEEE_comparisons=(ON|OFF)
    Default=OFF
    Enable/disable IEEE comparisons (ex: "x==x" for Nan and Inf).

–OPT:Olimit=n
    Stop optimizing after "n" threshold (0 means no limit).

–OPT:pad_common=(ON|OFF)
    Default=OFF
    Enable/disable common block padding (beware of DATA and EQUIVALENCE)

Specific Optimisation Options – 2

–OPT:unroll_times_max=*level*

Control the number of times a loop is unrolled (default is *level*=8 on R8K and R10K, 4 on other processors)

–OPT:unroll_size=*level*

Do not unroll the loop if the inner loop body contains more than *level* instructions (default is *level* = 80)

–OPT:unroll_bblimit=*level*

Do not unroll loops with more than *level* basic blocks (default is *level* =10)

Software Pipelining Control

–SWP:=(ON|OFF)
Default=OFF (–O0,1,2), ON (–O3)
Enable/disable SWP.

–SWP:trip_count_min=n
Default=5
Do not SWP loops with less than "n" iterations.

–SWP:unroll_times_max=n
Default=4 (r8000), 1 (other processors)
Unroll loops by "n" before SWP.

–SWP:memory_pairs_scheduling=(TRUE|FALSE)
Default=TRUE
Try to optimize odd/even bank accesses. FALSE may help find a better SWP schedule, introduced for the R8000.

Software Pipelining Control

–SWP:backtrack_max=n

Control how many attempts are made to SWP.

–SWP:body_ins_count_max=n

Default=100

SWP loops containing at most "n" instructions (0 means no limit).

–SWP:prune_prefetches=(TRUE|FALSE)

Default=TRUE (r10000); FALSE (other processors)

Enable/disable if prefetch operations in final schedule are replicated in light of stride and remove duplicates.

Code Generator

–CG:optimization_level=(0,1,2,3)

Used to override –O#.

–CG:if_conversion=(TRUE|FALSE)

Default=TRUE

Replace conditional branches by conditional move instructions.

```
do i=1,n
   if (a(i).lt.b(i) then
      c(i) = a(i)
   else
      c(i) = b(i)
   end if
end do
```

–CG:body_ins_count_max=n

Default=100

Set max number of instructions in body of loop to be if converted (0 means no limit).

–CG:prefetch=(TRUE|FALSE)

Default=FALSE

Control if prefetch operations are generated.

Global Code Motion

Redistribute instructions among basic blocks to improve instruction parallelism. In speculative mode instructions can be moved intodifferent control paths than the original program.

–GCM:prepass=(ON|OFF)

Enable/disable GCM pass before instruction scheduling and register allocation

–GCM:postpass=(ON|OFF)

Enable/disable GCM pass after instruction scheduling and register allocation

–GCM:aggressive_speculation=ON

Move instructions to basic blocks which are already using most of resources

–GCM:array_speculation=ON

Move top of loops to the bottom allowing array references past normal bounds.

–GCM:pointer_speculation=ON

Allow code motion which may yield invalid pointer references (i.e. NULL)

–GCM:static_load_speculation=ON

Allow motions of loads from static data areas

GCM operates in conjunction with exception handlers (–TENV:X=0..5)

Summary

What can *You* do?

Avoid equivalences

Avoid goto's

Keep commons consistent

Don't hand unroll

(unless you don't want the compiler to unroll)

Don't violate the Fortran standard

e.g., out–of–bounds indexing

What can *You* do?

For C code, use the most aggressive aliasing option possible

```
sub(a,b,c) {
...
for (i=0; i<n; i++) {
   for (j=0; j<n; j++) {
      for (k=0; k<n; k++) {
         c[i][j] += a[i][k]*b[k][j];
      }
   }
}
```

–OPT:alias=disjoint, –OPT:alias=restrict

Use ivdep directive where applicable

–OPT:liberal_ivdep, –OPT:cray_ivdep

Porting and Tuning Methodology

- **Start safe**
  –O0 –n32 –static
- **Get rid of the "static" flag (performance issue)**
  –O0 –n32

debug phase

- **Use the highest optimisation level**
  –O3 –mips4 –64/–n32
- **More aggressive optimisation**
  –O3 –mips4 –64/–n32
  –OPT:alias=restrict:roundoff=2:IEEE_arithmetic=3
- **"Give it all" (where you have to)**
  –O3 –mips4 –64/–n32 –OPT:... –LNO:... –IPA:.. –TARG:.... –pfa
- **"Living on the edge"**
  Start experimenting with some of the more esoteric options