Performance Programming with IBM pSeries Compilers and Libraries

Roch Archambault (archie@ca.ibm.com)

Agenda

- **Quick overview of compilers, libraries and tools**
- **Getting started**

  Installation, invocation
- **Common compiler controls**

  Language levels, environment, checking
- **Optimization controls**

  Optimization levels, target machine, profile feedback, link-time optimization
- **Directives and pragmas**
- **The MASS Library**

  Content, usage, performance and accuracy
- **Shared memory parallelism**

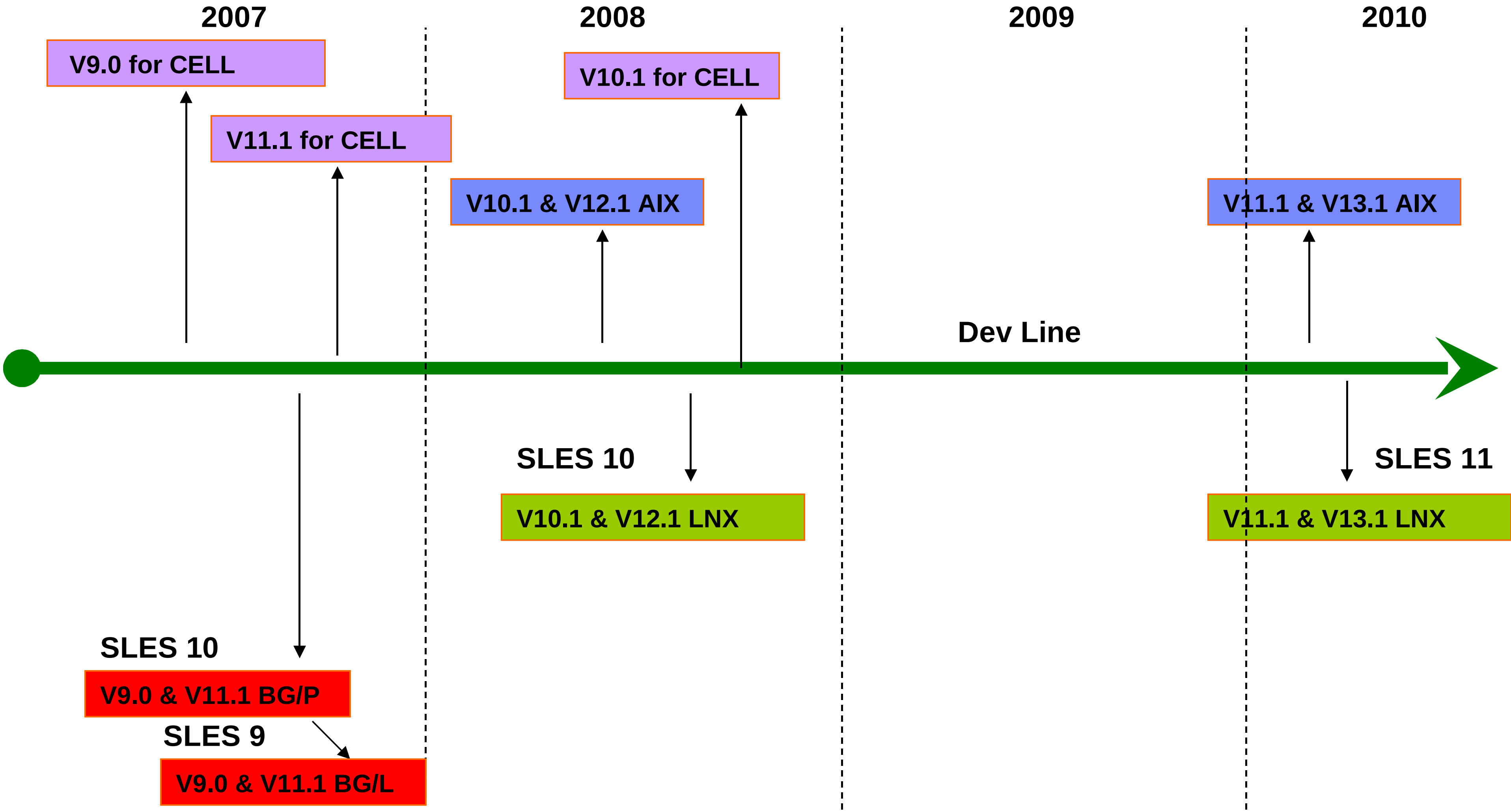
  Options, directives, environment
- **Performance programming**

  What can my compiler do for me?

  What can I do for my compiler?
- **VMX exploitation**
- **Programming for POWER6**

Overview of Compilers, Libraries and Tools

Roadmap of XL Compiler Releases



All information subject to change without notice

The System p Compiler Products: Previous Versions

- **All POWER4, POWER5, POWER5+ and PPC970 enabled**

  XL C/C++ Enterprise Edition V8.0 for AIX

  XL Fortran Enterprise Edition V10.1 for AIX

  XL C/C++ Advanced Edition V8.0 for Linux (SLES 9 & RHEL4)

  XL Fortran Advanced Edition V10.1 for Linux (SLES 9 & RHEL4)

  XL C/C++ Advanced Edition V8.0.1 for Linux (SLES 10 & RHEL4)

  XL Fortran Advanced Edition V10.1.1 for Linux (SLES 10 & RHEL4)

  XL C/C++ Enterprise Edition for AIX, V9.0 **(POWER6 enabled)**

  XL Fortran Enterprise Edition for AIX, V11.1 **(POWER6 enabled)**

  XL C/C++ Advanced Edition for Linux, V9.0 **(POWER6 enabled)**

  XL Fortran Advanced Edition for Linux, V11.1 **(POWER6 enabled)**

The System p Compiler Products: Latest Versions

- **All POWER4, POWER5, POWER5+ and PPC970 enabled**
  - XL C/C++ Enterprise Edition for AIX, V10.1 (July 2008)
  - XL Fortran Enterprise Edition for AIX, V12.1 (July 2008)
  - XL C/C++ Advanced Edition for Linux, V10.1 (September 2008)
  - XL Fortran Advanced Edition for Linux, V12.1 (September 2008)
- **Blue Gene (BG/L and BG/P) enabled**
  - XL C/C++ Advanced Edition for BG/L, V9.0
  - XL Fortran Advanced Edition for BG/L, V11.1
  - XL C/C++ Advanced Edition for BG/P, V9.0
  - XL Fortran Advanced Edition for BG/P, V11.1
- **Cell/B.E. cross compiler products:**
  - XL C/C++ for Multicore Acceleration for Linux on System p, V9.0
  - XL C/C++ for Multicore Acceleration for Linux on x86 Systems, V9.0
  - XL Fortran for Multicore Acceleration for Linux on System p, V11.1

The System p Compiler Products: Latest Versions

- **Technology Preview currently available from alphaWorks**

XL UPC language support on AIX and Linux

Download: **http://www.alphaworks.ibm.com/tech/upccompiler**

XL C/C++ for Transactional Memory for AIX

Download: **http://www.alphaworks.ibm.com/tech/xlcstm**

The System p Compiler Products: Future Versions

- Cell/B.E. cross compilers:
  - XL C/C++ for Multicore Acceleration for Linux on System p, V10.1 (4Q2008)
  - XL C/C++ for Multicore Acceleration for Linux on x86 Systems, V10.1 (4Q2008)
- POWER7 support
  - XL C/C++ Enterprise Edition for AIX, V11.1 (approx. 2010)
  - XL Fortran Enterprise Edition for AIX, V13.1 (approx 2010)
  - XL C/C++ Advanced Edition for Linux, V11.1 (approx 2010)
  - XL Fortran Advanced Edition for Linux, V13.1 (approx 2010)

All information subject to change without notice

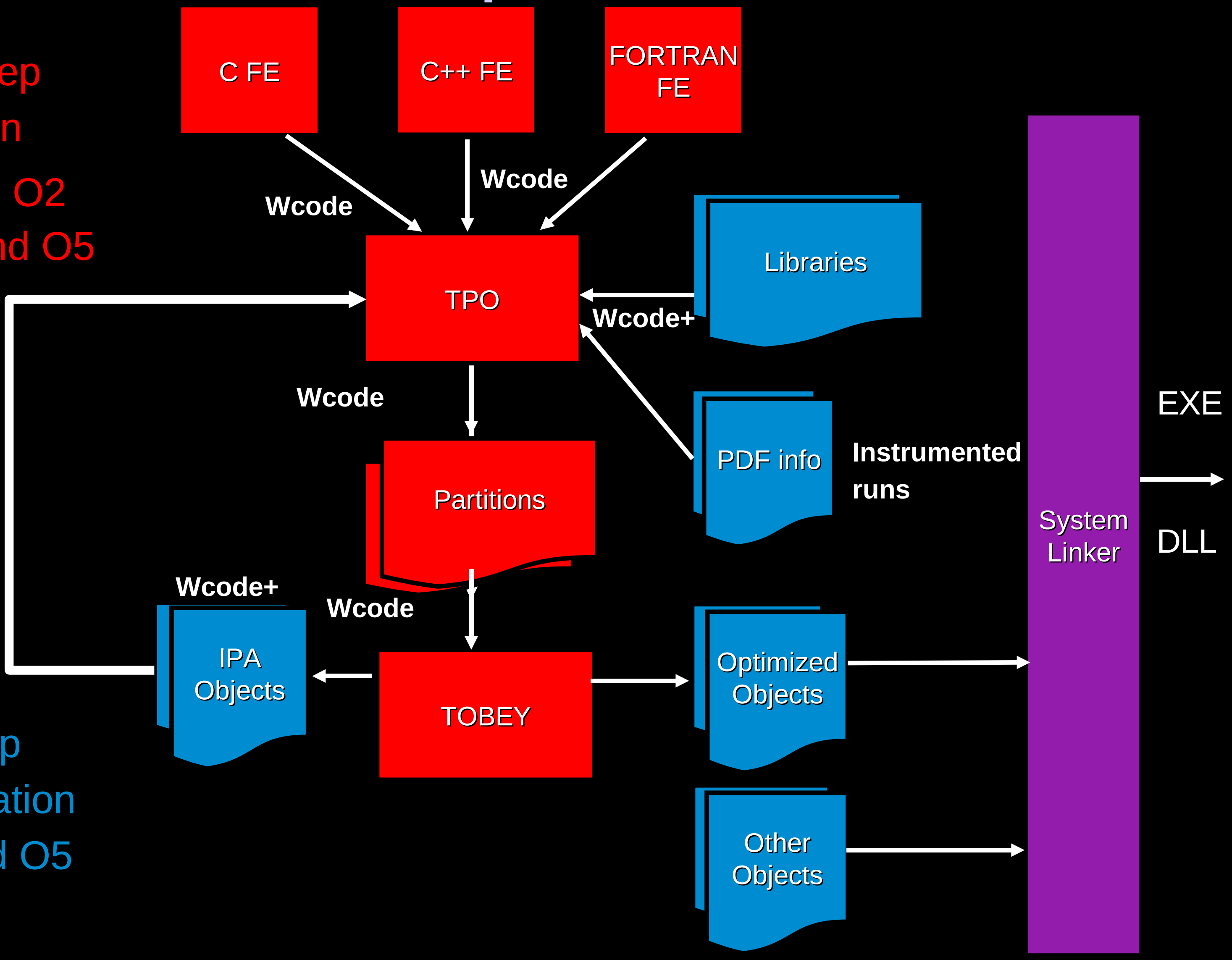
Common Fortran, C and C++ Features

- Linux (SLES and RHEL) and AIX, 32 and 64 bit
- Debug support
  - Debuggers on AIX:
    - Total View (TotalView Technologies), DDT (Allinea), IBM Debugger and DBX
  - Debuggers on Linux:
    - TotalView, DDT and GDB
- Full support for debugging of OpenMP programs (TotalView)
- Snapshot directive for debugging optimized code
- Portfolio of optimizing transformations
  - Instruction path length reduction
  - Whole program analysis
  - Loop optimization for parallelism, locality and instruction scheduling
  - Use profile directed feedback (PDF) in most optimizations
- Tuned performance on POWER3, POWER4, POWER5, PPC970, PPC440, POWER6 and CELL systems
- Optimized OpenMP

IBM XL Compiler Architecture

Compile Step
Optimization

noopt and O2
O3, O4 and O5



Link Step
Optimization
O4 and O5

XL Fortran Roadmap: Strategic Priorities

- **Superior Customer Service**

  Continue to work closely with key ISVs and customers in scientific and technical computing industries

- **Compliance to Language Standards and Industry Specifications**

  OpenMP API V2.5

  Fortran 77, 90 and 95 standards

  Fortran 2003 Standard

- **Exploitation of Hardware**

  Committed to maximum performance on POWER4, PPC970, POWER5, POWER6, PPC440, PPC450, CELL and successors

  Continue to work very closely with processor design teams

XL Fortran Version 12.1 for AIX/Linux – Summer/Fall 2008

New features since XL Fortran Version 10.1:

- Continued rollout of Fortran 2003
- Compliant to OpenMP V2.5
- Perform subset of loop transformations at –O3 optimization level
- Tuned BLAS routines (DGEMM and DGEMV) are included in compiler runtime (libxlopt)
- Recognize matrix multiply and replace with call to DGEMM
- Runtime check for availability of ESSL
- Support for auto-simdization and VMX intrinsics (and data types) on AIX
- Inline MASS library functions (math functions)
- Partial support for OpenMP V3.0
- Fine grain control for –qstrict option
- Improved compile/link time
- More Interprocedural data reorganization optimizations

XL C/C++ Roadmap: Strategic Priorities

- **Superior Customer Service**
- **Compliance to Language Standards and Industry Specifications**

ANSI / ISO C and C++ Standards

OpenMP API V3.0

- **Exploitation of Hardware**

Committed to maximum performance on POWER4, PPC970, POWER5, PPC440, POWER6, PPC450, CELL and successors

Continue to work very closely with processor design teams

- **Exploitation of OS and Middleware**

Synergies with operating system and middleware ISVs (performance, specialized function)

Committed to AIX Linux affinity strategy and to Linux on pSeries

- **Reduced Emphasis on Proprietary Tooling**

Affinity with GNU toolchain

XL C/C++ Version 10.1 for AIX/Linux – Summer/Fall 2008

New features since XL C/C++ Version 8.0:

- Exploit “restrict” keyword in C 1999
- Partial compliance to C++ TR1 libraries and Boost 1.34.0
- Support for -qtemplatedepth which allows the user to control number of recursive template instantiations allowed by the compiler.
- Exploit DFP and VMX on Power6.
- Improved inline assembler support
- Full support for OpenMP V3.0
- Fine grain control for –qstrict option
- Improved compile/link time
- More Interprocedural data reorganization optimizations

MASS (Math Acceleration SubSystem)

- Fast elementary functions
- Scalar and vector forms
- Tradeoff between accuracy and performance different than libm or compiler generated code
  - Accuracy differences small and usually tolerable
  - Vector function results does not depend on vector length
- Exceptions not always reported correctly
  - Some exceptions masked, some spuriously reported
- MASS functions assume round to nearest mode
- More later on performance and accuracy
- Included with XL Fortran V9.1 and XL C/C++ V7.0 compilers and subsequent releases
- More info: http://www.ibm.com/software/awdtools/mass/

ESSL (Engineering & Scientific Subroutine Library)

- Over 400 high-performance subroutines specifically tuned for pSeries and POWER4
- Parallel ESSL has over 100 high-performance subroutines designed for SP systems up to 512 nodes
- BLAS, ScaLAPACK and PBLAS compatibility
- Linear Algebraic Equations, Eigensystem Analysis, Fourier Transforms, Random Numbers
- More info:

  http://www.ibm.com/servers/eserver/pseries/library/sp_books/essl.html

Getting Started

Compiler commands

- The command used to invoke the compiler implies a predefined set of compiler options
- These can be controlled using the configuration file (default is /etc/xlf.cfg but you can write your own)
- Examples:

```
xlf –c a.f -g
```

Compiles F77 source in a.f and generates debugging information

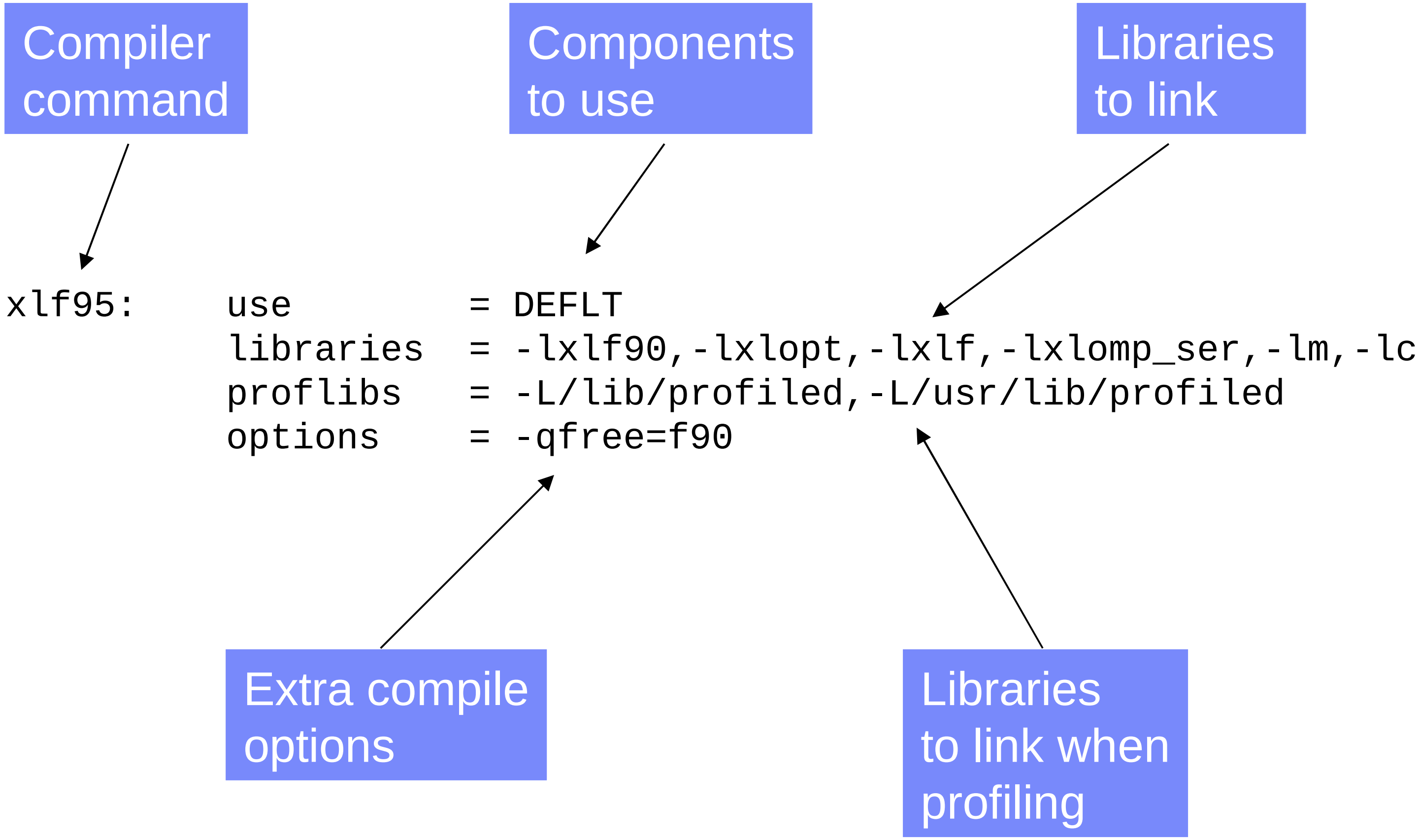
```
xlf90 –c –O2 b.f –qsuffix=f=f90 c.f90
```

Compiles F90 source in b.f and c.f90 with optimization level 2

```
xlf95_r -c d.f –qsmp
```

Compiles F95 source in d.f for SMP (compiler and libraries assume threaded code)

Example Configuration File Stanza



Compiler command

Components to use

Libraries to link

```
xlf95:    use        = DEFLT
          libraries  = -lxlf90,-lxlopt,-lxlf,-lxlomp_ser,-lm,-lc
          proflibs   = -L/lib/profiled,-L/usr/lib/profiled
          options    = -qfree=f90
```

Extra compile options

Libraries to link when profiling

Installation of Multiple Compiler Versions

- Installation of multiple compiler versions is supported
- The vacppndi and xlfndi scripts shipped with VisualAge C++ 6.0 and XL Fortran 8.1 and all subsequent releases allow the installation of a given compiler release or update into a non-default directory
- The configuration file can be used to direct compilation to a specific version of the compiler

  Example: xlf_v8r1 –c foo.f

  May direct compilation to use components in a non-default directory
- Care must be taken when multiple runtimes are installed on the same machine (details on next slide)

Coexistence of Multiple Compiler Runtimes

- **Backward compatibility**

  C, C++ and Fortran runtimes support backward compatibility.

  Executables generated by an earlier release of a compiler will work with a later version of the run-time environment.

- **Concurrent installation**

  Multiple versions of a compiler and runtime environment can be installed on the same machine

  Full support in xlfndi and vacppndi scripts is now available

- **Limited support for coexistence**

  LIBPATH must be used to ensure that a compatible runtime version is used with a given executable

  Only one runtime version can be used in a given process.

  Renaming a compiler library is not allowed.

  Take care in statically linking compiler libraries or in the use of *dlopen* or *load* .

  Details in the compiler FAQ **http://www.ibm.com/software/awdtools/fortran/xlfortran/support/**

  http://www.ibm.com/software/awdtools/xlcpp/support/

Common Compiler Controls

Controlling Language Level: Fortran

- Compiler invocations for standard compliant compilations
  - xlf or f77: Fortran 77
  - xlf90: Fortran 90
  - xlf95: Fortran 95
  - xlf2003: Fortran 2003 New!
- Finer control through -qlanglvl, -qxlf77 and –qxlf90 options
  - Slight tweaks to I/O behaviour
  - Intrinsic function behaviour
  - -qlanglvl can be used for additional diagnostics
- Non-standard language variations
  - -qautodbl to promote real types
  - -qintsize and -qrealsize to set default integer and real size
  - -qport for various extensions from other vendors

Controlling Language Level: C/C++

- **Compiler invocations for standard compliant compilations**
  - cc: “traditional” K&R C
  - xlc or c89: ANSI89 standard C
  - xlC: ANSI98 standard C++
  - c99: ANSI99 standard C
  - gxlc: “gcc-like” command line
  - gxlC: “g++-like” command line
- **Finer control through -qlanglvl**
  - strict conformance checking
  - lots of C++ language variations
  - gcc compatibility control
- **Non-standard language variations**
  - -qansialias, -qchars, -qcpluscmt, -qdollar, -qkeepinlines, -qkeyword, -qrtti

Common Environment Variables

- TMPDIR

  Redirect temporary disk storage used by the compiler
- OBJECT_MODE

  OBJECT_MODE=32 or OBJECT_MODE=64 supported
- LANG

  Specifies language to use for messages
- NLSPATH

  Specified search path for messages – useful in non-default installations
- XLFRTEOPTS

  Tweak Fortran runtime behaviour
- XLSMPOPTS

  Tweak SMP runtime behaviour

Program Checking

- -qcheck

  In Fortran, does bounds checking on array references, array sections and character substrings

  In C/C++, checks for NULL pointers, for divide by zero and for array indices out of bounds
- -qextchk, -btypchk

  Generates type hash codes so that the AIX linker can check type consistency across files (also done by -qipa)
- -qinitauto

  Generates extra code to initialize stack storage

  Can be done bytewise or wordwise

Program Behaviour Options (-qstrict)

- -q[no]strict
  - Default is -qstrict with -qnoopt and -O2, -qnostrict with -O3, -O4, -O5
  - -qnostrict allows the compiler to reorder floating point calculations and potentially excepting instructions
  - Use -qstrict when your computation legitimately involves NaN, INF or denormalized values
  - Use -qstrict when exact compatibility is required with another IEEE compliant system
  - Note that -qstrict disables many potent optimizations so use it only when necessary and consider applying it at a file or even function level to limit the negative impact

Program Behaviour Options (Aliasing)

- -qalias (Fortran)

  Specified as -qalias=[no]std:[no]aryovrlp:others

  Allows the compiler to assume that certain variables do not refer to overlapping storage

  std (default) refers to the rule about storage association of reference parameters with each other and globals

  aryovrlp (default) defines whether there are any assignments between storage-associated arrays - try -qalias=noaryovrlp for better performance your Fortran 90 code has no storage associated assignments

Program Behaviour Options (Aliasing)

- -qalias (C, C++)

  Similar to Fortran option of the same name but focussed on overlap of storage accessed using pointers

  Specified as -qalias=subopt where subopt is one of:

  [no]ansi: Enable ANSI standard type-based alias rules (ansi is default when using "xlc", noansi is default when using "cc")

  [no]typeptr: Assume pointers to different types never point to the same or overlapping storage - use if your pointer usage follows strict type rules

  [no]allptrs: Assume that different pointer variables always point to non-overlapping storage - use only in selected situations where pointers never overlap

  [no]addrtaken: Assume that external variables do not have their address taken outside the source file being compiled

Why the big fuss about aliasing?

- The precision of compiler analyses is gated in large part by the apparent effects of direct or indirect memory writes and the apparent presence of direct or indirect memory reads.
- Memory can be referenced directly through a named symbol, indirectly through a pointer or reference parameter, or indirectly through a function call.
- Many apparent references to memory are false and these constitute barriers to compiler analysis.
- The compiler does analysis of possible aliases at all optimization levels but analysis of these apparent references is best when using -qipa since it can see through most calls.
- Options such as -qalias and directives such as disjoint, isolated_call, CNCALL, PERMUTATION and INDEPENDENT can have pervasive effect since they fundamentally improve the precision of compiler analysis.

Floating Point Control (-qfloat)

- Precise control over the handling of floating point calculations
- Defaults are almost IEEE 754 compliant
- Specified as -qfloat=subopt where subopt is one of:

[no]fold: enable compile time evaluation of floating point calculations - may want to disable for handling of certain exceptions (eg. overflow, inexact)

[no]maf: enable generation of multiple-add type instructions - may want to disable for exact compatibility with other machines but this will come at a high price in performance

[no]rrm: specifies that rounding mode may not be round-to-nearest (default is norrm) or may change across calls

[no]rsqrt: allow computation of a divide by square root to be replaced by a multiply of the reciprocal square root

Floating Point Control (-qfloat)

- **New –qfloat suboptions added in XL Fortran V11.1 and XL C/C++ V9.0**

New!

[no]fenv:	asserts that FPSCR may be accessed (default is nofenv)
[no]hscmplx	better performance for complex divide/abs (defaults is nohscmplx)
[no]single	does not generate single precision float operations (default is single)
[no]rngchk	does not generate range check for software divide (default is rngchk)

New!

- **-qxlf90=nosignedzero now the default when –qnostrict**
  improves max/min performance by generating fsel instruction instead of branch sequence

Floating Point Trapping (-qflttrap)

- Enables software checking of IEEE floating point exceptions
- Usually more efficient than hardware checking since checks can be executed less frequently
- Specified as -qflttrap=imprecise | enable | ieee_exceptions

  -qflttrap=imprecise: check for error conditions at procedure entry/exit, otherwise check after any potentially excepting instruction

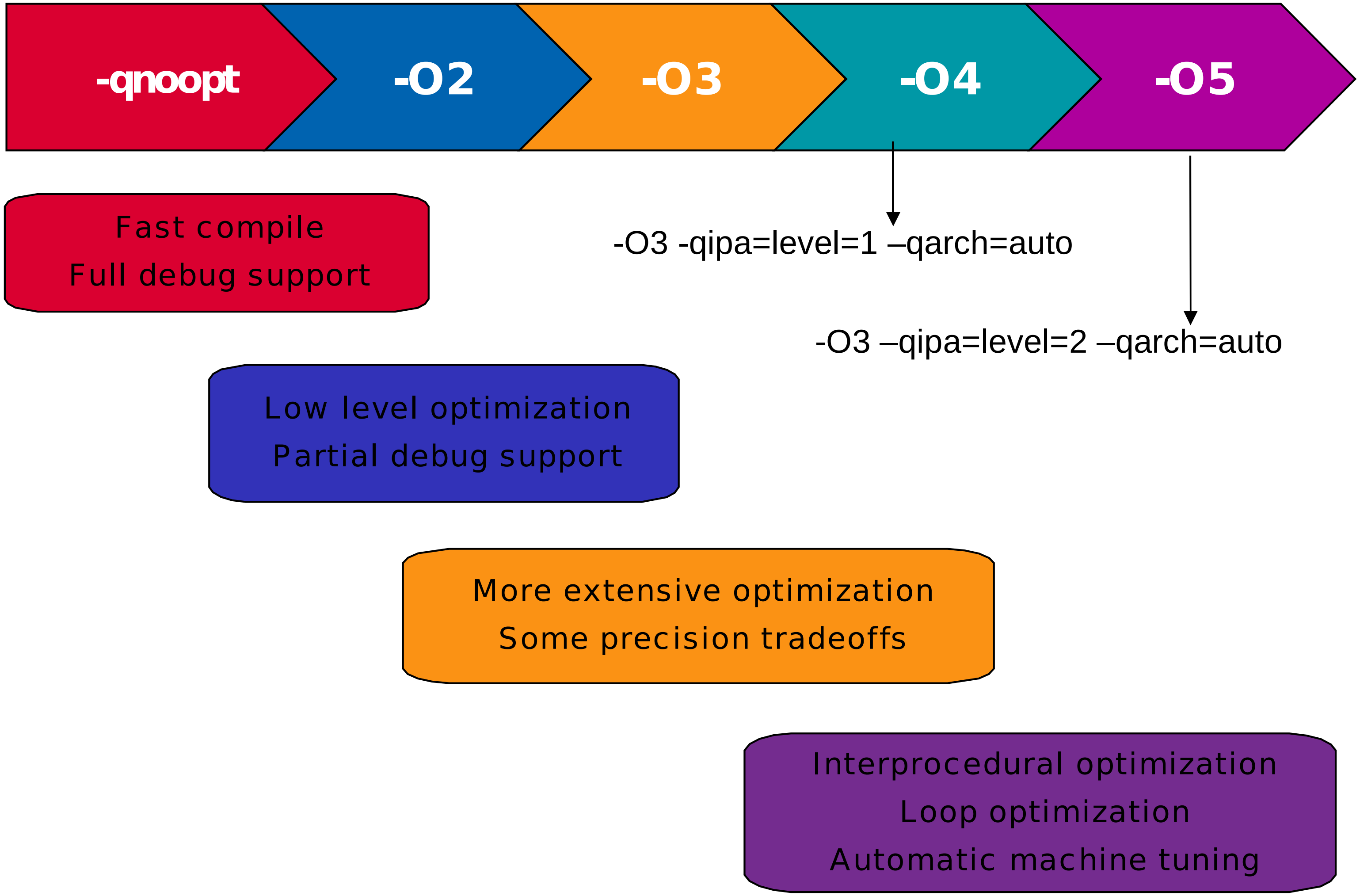
  -qflttrap=enable: enables generation of checking code, also enables exceptions in hardware

  -qflttrap=overflow:underflow:zerodivide:inexact: check given conditions
- In the event of an error, SIGTRAP is raised

  As a convenience the -qsigtrap option will install a default handler which dumps a stack trace at the point of error (Fortran only)

Optimization Controls

Optimization Levels



-qnoopt

-O2

-O3

-O4

-O5

-O3 -qipa=level=1 –qarch=auto

-O3 –qipa=level=2 –qarch=auto

Fast compile
Full debug support

Low level optimization
Partial debug support

More extensive optimization
Some precision tradeoffs

Interprocedural optimization
Loop optimization
Automatic machine tuning

Example: Matrix Multiply

```
DO I = 1, N1
  DO J = 1, N3
    DO K = 1, N2
      C(I,J) = C(I,J) + A(K,I) * B(J,K)
    END DO
  END DO
END DO
```

Matrix Multiply inner loop code with -qnoopt

38 instructions, 31.4 cycles per iteration

```
__L1:
    lwz     r3,160(SP)
    lwz     r9,STATIC_BSS
    lwz     r4,24(r9)
    subfi   r5,r4,-8
    lwz     r11,40(r9)
    mullw   r6,r4,r11
    lwz     r4,36(r9)
    rlwinm  r4,r4,3,0,28
    add     r7,r5,r6
    add     r7,r4,r7
    lfdx    fp1,r3,r7
    lwz     r7,152(SP)
    lwz     r12,0(r9)
    subfi   r10,r12,-8
    lwz     r8,44(r9)
    mullw   r12,r12,r8
    add     r10,r10,r12
    add     r10,r4,r10
    lfdx    fp2,r7,r10
    lwz     r7,156(SP)
    lwz     r10,12(r9)
    subfi   r9,r10,-8
    mullw   r10,r10,r11
    rlwinm  r8,r8,3,0,28
    add     r9,r9,r10
    add     r8,r8,r9
    lfdx    fp3,r7,r8
    fmadd   fp1,fp2,fp3,fp1
    add     r5,r5,r6
    add     r4,r4,r5
    stfdx   fp1,r3,r4
    lwz     r4,STATIC_BSS
    lwz     r3,44(r4)
    addi    r3,1(r3)
    stw     r3,44(r4)
    lwz     r3,112(SP)
    addic.  r3,r3,-1
    stw     r3,112(SP)
    bgt     __L1
```

Matrix Multiply inner loop code with -qnoopt

necessary instructions

```
__L1:
    lwz     r3,160(SP)
    lwz     r9,STATIC_BSS
    lwz     r4,24(r9)
    subfi   r5,r4,-8
    lwz     r11,40(r9)
    mullw   r6,r4,r11
    lwz     r4,36(r9)
    rlwinm  r4,r4,3,0,28
    add     r7,r5,r6
    add     r7,r4,r7
    lfdx    fp1,r3,r7
    lwz     r7,152(SP)
    lwz     r12,0(r9)
    subfi   r10,r12,-8
    lwz     r8,44(r9)
    mullw   r12,r12,r8
    add     r10,r10,r12
    add     r10,r4,r10
    lfdx    fp2,r7,r10
```

```
    lwz     r7,156(SP)
    lwz     r10,12(r9)
    subfi   r9,r10,-8
    mullw   r10,r10,r11
    rlwinm  r8,r8,3,0,28
    add     r9,r9,r10
    add     r8,r8,r9
    lfdx    fp3,r7,r8
    fmadd   fp1,fp2,fp3,fp1
    add     r5,r5,r6
    add     r4,r4,r5
    stfdx   fp1,r3,r4
    lwz     r4,STATIC_BSS
    lwz     r3,44(r4)
    addi    r3,1(r3)
    stw     r3,44(r4)
    lwz     r3,112(SP)
    addic.  r3,r3,-1
    stw     r3,112(SP)
    bgt     __L1
```

Matrix Multiply inner loop code with -qnoopt

necessary instructions   loop control

```
__L1:
    lwz     r3,160(SP)
    lwz     r9,STATIC_BSS
    lwz     r4,24(r9)
    subfi   r5,r4,-8
    lwz     r11,40(r9)
    mullw   r6,r4,r11
    lwz     r4,36(r9)
    rlwinm  r4,r4,3,0,28
    add     r7,r5,r6
    add     r7,r4,r7
    lfdx    fp1,r3,r7
    lwz     r7,152(SP)
    lwz     r12,0(r9)
    subfi   r10,r12,-8
    lwz     r8,44(r9)
    mullw   r12,r12,r8
    add     r10,r10,r12
    add     r10,r4,r10
    lfdx    fp2,r7,r10
    lwz     r7,156(SP)
    lwz     r10,12(r9)
    subfi   r9,r10,-8
    mullw   r10,r10,r11
    rlwinm  r8,r8,3,0,28
    add     r9,r9,r10
    add     r8,r8,r9
    lfdx    fp3,r7,r8
    fmadd   fp1,fp2,fp3,fp1
    add     r5,r5,r6
    add     r4,r4,r5
    stfdx   fp1,r3,r4
    lwz     r4,STATIC_BSS
    lwz     r3,44(r4)
    addi    r3,1(r3)
    stw     r3,44(r4)
    lwz     r3,112(SP)
    addic.  r3,r3,-1
    stw     r3,112(SP)
    bgt     __L1
```

Matrix Multiply inner loop code with -qnoopt

necessary instructions   loop control   addressing code

```
__L1:
    lwz     r3,160(SP)
    lwz     r9,STATIC_BSS
    lwz     r4,24(r9)
    subfi   r5,r4,-8
    lwz     r11,40(r9)
    mullw   r6,r4,r11
    lwz     r4,36(r9)
    rlwinm  r4,r4,3,0,28
    add     r7,r5,r6
    add     r7,r4,r7
    lfdx    fp1,r3,r7
    lwz     r7,152(SP)
    lwz     r12,0(r9)
    subfi   r10,r12,-8
    lwz     r8,44(r9)
    mullw   r12,r12,r8
    add     r10,r10,r12
    add     r10,r4,r10
    lfdx    fp2,r7,r10
```

```
lwz     r7,156(SP)
lwz     r10,12(r9)
subfi   r9,r10,-8
mullw   r10,r10,r11
rlwinm  r8,r8,3,0,28
add     r9,r9,r10
add     r8,r8,r9
lfdx    fp3,r7,r8
fmadd   fp1,fp2,fp3,fp1
add     r5,r5,r6
add     r4,r4,r5
stfdx   fp1,r3,r4
lwz     r4,STATIC_BSS
lwz     r3,44(r4)
addi    r3,1(r3)
stw     r3,44(r4)
lwz     r3,112(SP)
addic.  r3,r3,-1
stw     r3,112(SP)
bgt     __L1
```

Optimization Level –O2 (same as –O)

- Comprehensive low-level optimization
  - Global assignment of user variables to registers
  - Strength reduction and effective usage of addressing modes
  - Elimination of unused or redundant code
  - Movement of invariant code out of loops
  - Scheduling of instructions for the target machine
  - Some loop unrolling and pipelining
- Partial support for debugging
  - Externals and parameter registers visible at procedure boundaries
  - Snapshot pragma/directive creates additional program points for storage visibility
  - -qkeepparm option forces parameters to memory on entry so that they can be visible in a stack trace

Matrix Multiply Inner Loop with –O2

load/store of "C"
moved out of loop

```
   lfdux  fp0,r12,r8
__L1:
   lfdux  fp1,r31,r7
   lfdu   fp2,8(r30)
   fmadd  fp0,fp1,fp2,fp0
   bdnz   __L1
   stfd   fp0,0(r12)
```

strength reduction
update-form loads

hardware assisted
loop control

3 instructions, 3.1 cycles per iteration

Matrix Multiply Inner Loop with –O2 –qtune=pwr4

```
   lfdux   fp2,r31,r7
   lfdu    fp1,8(r30)
   bdz     __L2
__L1:
   fmadd   fp0,fp2,fp1,fp0
   lfdux   fp2,r31,r7
   lfdu    fp1,8(r30)
   bdnz    __L1
__L2:
   fmadd   fp0,fp2,fp1,fp0
```

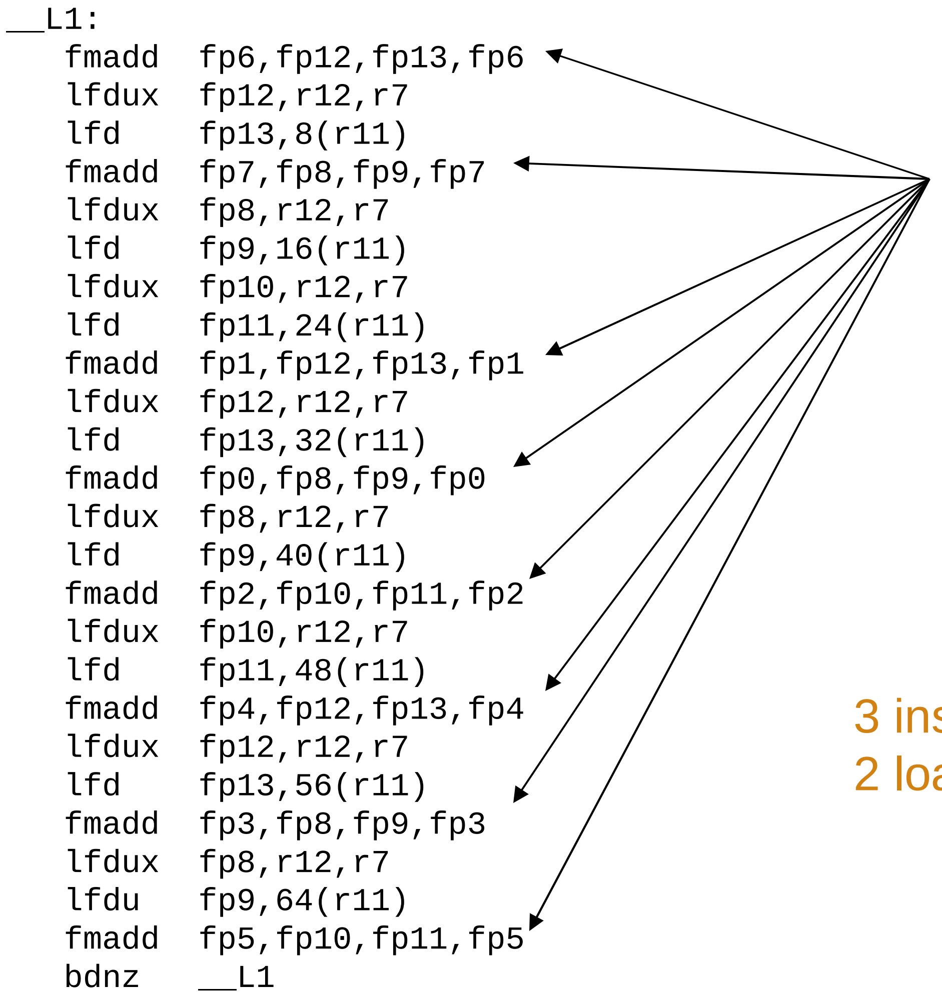
pipelined
execution

Optimization Level –O3

New!

- More extensive optimization
  - Deeper inner loop unrolling
  - Loop nest optimizations such as unroll-and-jam and interchange (-qhot subset)
  - Better loop scheduling
  - Additional optimizations allowed by -qnostrict
  - Widened optimization scope (typically whole procedure)
  - No implicit memory usage limits (-qmaxmem=-1)
- Some precision tradeoffs
  - Reordering of floating point computations
  - Reordering or elimination of possible exceptions (eg. divide by zero, overflow)

Matrix Multiply inner loop code with -O3 -qtune=pwr4



```
__L1:
   fmadd   fp6,fp12,fp13,fp6
   lfdux   fp12,r12,r7
   lfd     fp13,8(r11)
   fmadd   fp7,fp8,fp9,fp7
   lfdux   fp8,r12,r7
   lfd     fp9,16(r11)
   lfdux   fp10,r12,r7
   lfd     fp11,24(r11)
   fmadd   fp1,fp12,fp13,fp1
   lfdux   fp12,r12,r7
   lfd     fp13,32(r11)
   fmadd   fp0,fp8,fp9,fp0
   lfdux   fp8,r12,r7
   lfd     fp9,40(r11)
   fmadd   fp2,fp10,fp11,fp2
   lfdux   fp10,r12,r7
   lfd     fp11,48(r11)
   fmadd   fp4,fp12,fp13,fp4
   lfdux   fp12,r12,r7
   lfd     fp13,56(r11)
   fmadd   fp3,fp8,fp9,fp3
   lfdux   fp8,r12,r7
   lfdu    fp9,64(r11)
   fmadd   fp5,fp10,fp11,fp5
   bdnz    __L1
```

unrolled by 8

dot product accumulated in
8 interleaved parts (fp0-fp7)
(combined after loop)

3 instructions, 1.6 cycles per iteration
2 loads and 1 fmadd per iteration

Tips for getting the most out of –O2 and –O3

- If possible, test and debug your code without optimization before using -O2
- Ensure that your code is standard-compliant. Optimizers are the ultimate conformance test!
- In Fortran code, ensure that subroutine parameters comply with aliasing rules
- In C code, ensure that pointer use follows type restrictions (generic pointers should be char* or void*)
- Ensure all shared variables and pointers to same are marked volatile
- Compile as much of your code as possible with -O2.
- If you encounter problems with -O2, consider using -qalias=noansi or -qalias=nostd rather than turning off optimization.
- Next, use -O3 on as much code as possible.

New!

- If you encounter problems or performance degradations, consider using –qstrict, -qcompact, or -qnohot along with -O3 where necessary.
- If you still have problems with -O3, switch to -O2 for a subset of files/subroutines but consider using -qmaxmem=-1 and/or -qnostrict.

High Order Transformations (-qhot)

- Supported for all languages
- Specified as -qhot[=[no]vector | arraypad[=n] | [no]simd]
- Optimized handling of F90 array language constructs (elimination of temporaries, fusion of statements)
- High level transformation (eg. interchange, fusion, unrolling) of loop nests to optimize:
  - memory locality (reduce cache/TLB misses)
  - usage of hardware prefetch
  - loop computation balance (typically ld/st vs. float)
- Optionally transforms loops to exploit MASS vector library (eg. reciprocal, sqrt, trig) - may result in slightly different rounding
- Optionally introduces array padding under user control - potentially unsafe if not applied uniformly
- Optionally transforms loops to exploit VMX unit when –qarch=ppc970 or –qarch=pwr6

New!

Matrix multiply inner loop code with -O3 -qhot -qtune=pwr4

```
__L1:
   fmadd    fp1,fp4,fp2,fp1
   fmadd    fp0,fp3,fp5,fp0
   lfdux    fp2,r29,r9
   lfdu     fp4,32(r30)
   fmadd    fp10,fp7,fp28,fp10
   fmadd    fp7,fp9,fp7,fp8
   lfdux    fp26,r27,r9
   lfd      fp25,8(r29)
   fmadd    fp31,fp30,fp27,fp31
   fmadd    fp6,fp11,fp30,fp6
   lfd      fp5,8(r27)
   lfd      fp8,16(r28)
   fmadd    fp30,fp4,fp28,fp29
   fmadd    fp12,fp13,fp11,fp12
   lfd      fp3,8(r30)
   lfd      fp11,8(r28)
   fmadd    fp1,fp4,fp9,fp1
   fmadd    fp0,fp13,fp27,fp0
   lfd      fp4,16(r30)
   lfd      fp13,24(r30)
   fmadd    fp10,fp8,fp25,fp10
   fmadd    fp8,fp2,fp8,fp7
   lfdux    fp9,r29,r9
   lfdu     fp7,32(r28)
   fmadd    fp31,fp11,fp5,fp31
   fmadd    fp6,fp26,fp11,fp6
   lfdux    fp11,r27,r9
   lfd      fp28,8(r29)
   fmadd    fp12,fp3,fp26,fp12
   fmadd    fp29,fp4,fp25,fp30
   lfd      fp30,-8(r28)
   lfd      fp27,8(r27)
   bdnz     __L1
```

unroll-and-jam 2x2
inner unroll by 4
interchange "i" and "j" loops

2 instructions, 1.0 cycles per iteration
balanced: 1 load and 1 fmadd per iteration

With XLF V9.1 (-O3 -qhot –qarch=pwr4)

```
__L1:
  fmadd     fp15=fp10,fp6,fp3
  fmadd     fp9=fp9,fp5,fp3
  lfd       fp10=@TILEA0(gr27,-6168)
  addi      gr27=gr27,16
  fmadd     fp11=fp11,fp8,fp3
  fmadd     fp16=fp12,fp7,fp3
  lfd       fp24=a[](gr26,-23992)
  lfd       fp23=a[](gr26,-15992)
  fmadd     fp31=fp31,fp6,fp4
  fmadd     fp3=fp13,fp5,fp4
  lfd       fp22=a[](gr26,-7992)
  lfd       fp21=a[](gr26,8)
  fmadd     fp30=fp30,fp8,fp4
  fmadd     fp29=fp29,fp7,fp4
  lfd       fp12=@TILEA0(gr27,-4120)
  fmadd     fp17=fp27,fp6,fp19
  fmadd     fp28=fp28,fp5,fp19
  fmadd     fp25=fp25,fp8,fp19
  fmadd     fp4=fp26,fp7,fp19
  lfd       fp27=@TILEA0(gr27,-2056)
  fmadd     fp18=fp2,fp10,fp24
  fmadd     fp1=fp1,fp10,fp23
  fmadd     fp0=fp0,fp10,fp21
  fmadd     fp20=fp20,fp10,fp22
  lfd       fp2=@TILEA0(gr27,8)
  addi      gr26=gr26,16
  fmadd     fp9=fp9,fp24,fp12
  fmadd     fp10=fp15,fp23,fp12
```

Unroll-and-Jam 4x4

Inner unroll by 2

Interchange “i” and “j” loops

Tile “i” and “j” loops

Transpose blocks of b array

32 iterations in 20 cycles

```
lfd       fp19=@TILEA0(gr27,-6176)
fmadd     fp11=fp11,fp21,fp12
lfd       fp5=a[](gr26,-24000)
lfd       fp6=a[](gr26,-16000)
fmadd     fp12=fp16,fp22,fp12
fmadd     fp13=fp3,fp24,fp27
fmadd     fp31=fp31,fp23,fp27
lfd       fp7=a[](gr26,-8000)
lfd       fp8=a[](gr26,0)
fmadd     fp30=fp30,fp21,fp27
fmadd     fp29=fp29,fp22,fp27
lfd       fp3=@TILEA0(gr27,-4112)
fmadd     fp28=fp28,fp24,fp2
fmadd     fp27=fp17,fp23,fp2
fmadd     fp25=fp25,fp21,fp2
fmadd     fp26=fp4,fp22,fp2
lfd       fp4=@TILEA0(gr27,-2048)
fmadd     fp1=fp1,fp19,fp6
fmadd     fp2=fp18,fp19,fp5
fmadd     fp0=fp0,fp19,fp8
fmadd     fp20=fp20,fp19,fp7
lfd       fp19=@TILEA0(gr27,16)
bndz      __L1
```

Matmul Idiom Recognition

- In V8/10.1, we shipped some matmul functions in libxlopt
  - Users could insert explicit calls to these routines in their code
  - The libxlopt versions would automatically call the equivalent ESSL functions if ESSL was installed on the system.

New!

- In V9/11.1, the compiler recognizes some limited loop nest patterns as matrix multiplies and automatically generates calls to matmul functions in libxlopt or ESSL.
  - sgemm, dgemm

- The loop nest can be interchanged in any order, example next slide

Matmul Idiom Recognition (cont.)

```
subroutine foo(M,N,L,A,B,C,D)

   ...
   do i=1,M
     do j=1,N
       do k=1,L
          C(i,j) = C(i,j) + A(i,k)*B(k,j)
      end do
     end do
   end do

   do i=1,M
     do k=1,L
      do j=1,N
         D(i,j) = D(i,j) + A(i,k)*B(k,j)
      end do
     end do
   end do

   return
   end
```

```
.foo:
        mfspr   r0,LR
        stfd    fp31,-8(SP)
        stfd    fp30,-16(SP)
        st      r31,-20(SP)
        ....
        lfs     fp31,0(r11)
        stfd    fp31,136(SP)
        stfd    fp31,144(SP)
        bl      .dgemm{PR}
        ....
        st      r26,68(SP)
        st      r0,72(SP)
        l       r9,172(SP)
        bl      .dgemm{PR}
        oril    r0,r0,0x0000
        l       r12,248(SP)
        lfd     fp31,232(SP)
        ...
```

Tips for getting the most out of -qhot

- Try using -qhot along with -O2 or -O3 for all of your code. It is designed to have neutral effect when no opportunities exist.
- If you encounter unacceptably long compile times (this can happen with complex loop nests) or if your performance degrades with the use of -qhot, try using -qhot=novector, or -qstrict or -qcompact along with -qhot.
- If necessary, deactivate -qhot selectively, allowing it to improve some of your code.
- Read the transformation report generated using –qreport. If your hot loops are not transformed as you expect, try using assertive directives such as INDEPENDENT or CNCALL or prescriptive directives such as UNROLL or PREFETCH.
- When –qarch=ppc970, the default with –qhot is to perform SIMD-vectorization. You can specify –qhot=nosimd to disable SIMD-vectorization
- New with Fortran V10.1 and C/C++ V8.0:

New!

support –qhot=level=x where x is 0 or 1. Default is –qhot=level=1 when –qhot is specified.

-qhot=level=0 is the default when –O3 is specified.

Link-time Optimization (-qipa)

- Supported for all languages
- Can be specified on the compile step only or on both compile and link steps ("whole program" mode)
- Whole program mode expands the scope of optimization to an entire program unit (executable or shared object)
- Specified as -qipa[=level=n | inline= | fine tuning]

  level=0: Program partitioning and simple interprocedural optimization

  level=1: Inlining and global data mapping

  level=2: Global alias analysis, specialization, interprocedural data flow

  inline=: Precise user control of inlining

  fine tuning: Specify library code behaviour, tune program partitioning, read commands from a file

Tips for getting the most out of -qipa

- When specifying optimization options in a makefile, remember to use the compiler driver (cc, xlf, etc) to link and repeat all options on the link step:

```
LD = xlf

OPT = -O3 -qipa

FFLAGS=...$(OPT)...

LDFLAGS=...$(OPT)...
```

- -qipa works when building executables or shared objects but always compile 'main' and exported functions with -qipa.
- It is not necessary to compile everything with -qipa but try to apply it to as much of your program as possible.

More -qipa tips

- When compiling and linking separately, use -qipa=noobject on the compile step for faster compilation.
- Ensure there is enough space in /tmp (at least 200MB) or use the TMPDIR variable to specify a different directory.
- The "level" suboption is a throttle. Try varying the "level" suboption if link time is too long. -qipa=level=0 can be very beneficial for little cost.
- Look at the generated code. If too few or too many functions are inlined, consider using -qipa=[no]inline

Target Machines

- -qarch

  Specifies the target machine or machine family on which the generated program is expected to run successfully

  -qarch=ppc targets any PowerPC (default with XLF V11.1)

  -qarch=pwr4 targets POWER4 specifically

  -qarch=auto targets the same type of machine as the compiling machine

- -qtune

  Specifies the target machine on which the generated code should run best

  Orthogonal to –qarch setting but some combinations not allowed

  -qtune=pwr4 tunes generated code for POWER4 machines

  -qtune=auto tunes generated code to run well on machines similar to the compiling machine

  New!

  -qtune=balanced tunes generated code to run well on POWER5 and POWER6

  (Default with XLF V11.1)

Getting the most out of target machine options

- Try to specify with -qarch the smallest family of machines possible that will be expected to run your code correctly.
  - -qarch=ppc is better if you don't need to run on Power or Power2 but this will inhibit generation of sqrt or fsel, for example
  - -qarch=ppcgr is a bit better, since it allows generation of fsel but still no sqrt
  - To get sqrt, you will need -qarch=pwr3. This will also generate correct code for Power 4.
- Try to specify with -qtune the machine where performance should be best.
  - If you are not sure, try -qtune=balanced. This will generate code that should generally run well on most machines.

The –O4 and –O5 Options

- Optimization levels 4 and 5 automatically activate several other optimization options as a package
- Optimization level 4 (-O4) includes:

  -O3

  -qhot

  -qipa

  -qarch=auto

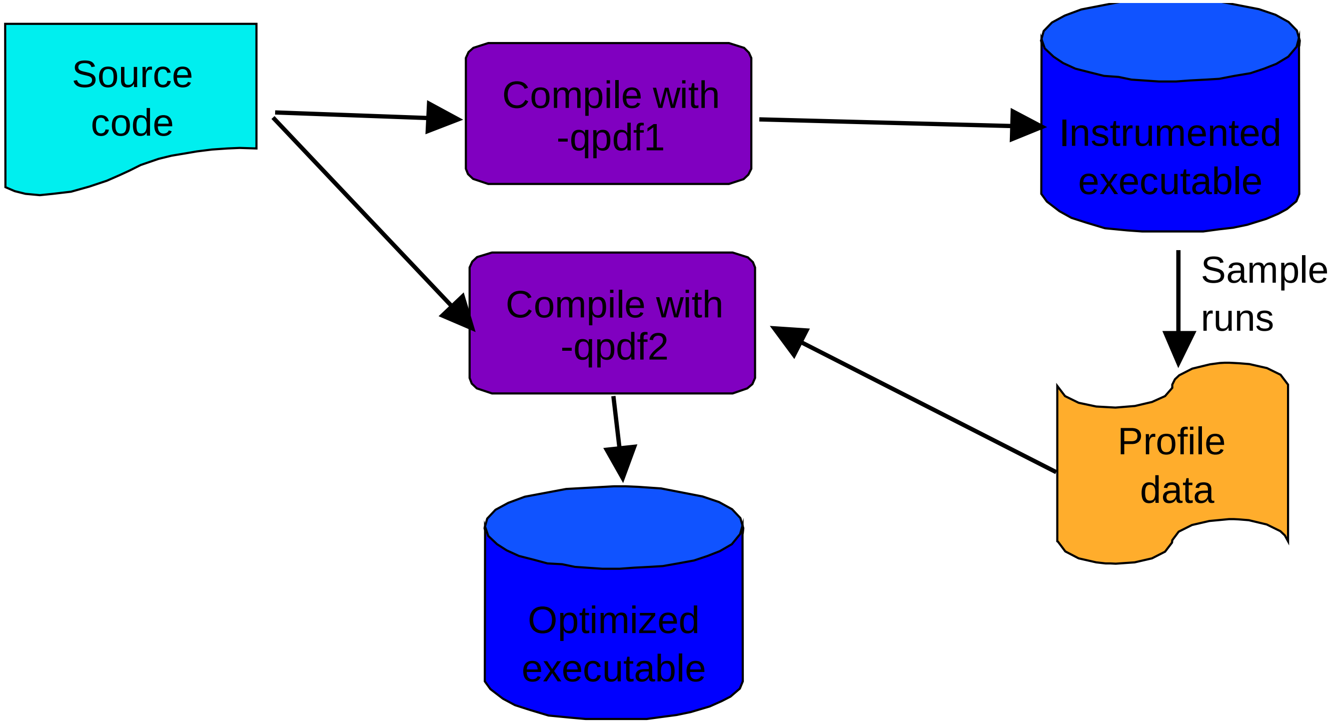
  -qtune=auto

  -qcache=auto
- Optimization level 5 (-O5) includes everything from -O4 plus:

  -qipa=level=2

Profile Feedback

- Profile directed feedback (PDF) is a two-stage compilation process that allows the user to provide additional detail about typical program behaviour to the compiler.



Profile Directed Feedback: Details

- Stage 1 is a regular compilation (using an arbitrary set of optimization options) with the -qpdf1 option added.
  - the resulting object code is instrumented for the collection of program control flow and other data
- The executable or shared object created by stage 1 can be run in a number of different scenarios for an arbitrary amount of time
- Stage 2 is a recompilation (only relinking is necessary with Fortran 8.1.1 or C/C++ 6.0) using exactly the same options except -qpdf2 is used instead of -qpdf1.
  - the compiler consumes previously collected data for the purpose of path-biased optimization
  - code layout, scheduling, register allocation
  - inlining decisions, partially invariant code motion, switch code generation, loop optimizations
- PDF should be used mainly on code which has rarely executed conditional error handling or instrumentation
- PDF usually has a neutral effect in the absence of firm profile information (ie. when sample data is inconclusive)
- However, always use characteristic data for profiling. If sufficient data is unavailable, do not use PDF.
- The –qshowpdf options and showpdf tool can be used to view the PDF information accumulated from the –qpdf1 run.

Miscellaneous Performance Options

- -qcompact: specified as -q[no]compact

  Prefers final code size reduction over execution time performance when a choice is necessary

  Can be useful as a way to constrain -O3 optimization

- -qinline: specified as -qinline[+names | -names] or -qnoinline

  Controls inlining of named functions - usable at compile time and/or link time

  Synonymous with -qipa=inline and -Q

- -qunroll: specified as -q[no]unroll

  Independently controls loop unrolling (implicitly activated under -O2 and -O3)

Miscellaneous Performance Options

- -qinlglue: specified as -q[no]inlglue

  Inline calls to "glue" code used in calls through function pointers (including virtual) and calls to functions which are dynamically bound

  Pointer glue is inlined by default for -qtune=pwr4

- -qtbtable

  Controls the generation of traceback table information:

  -qtbtable=none inhibits generation of tables - no stack unwinding is possible

  -qtbtable=small generates tables which allow stack unwinding but omit name and parameter information - useful for optimized C++

    This is the default setting when using optimization

  -qtbtable=full generates full tables including name and parameter information - useful for debugging

Miscellaneous Performance Options

- -q[no]eh (C++ only)

  Asserts that no throw is reachable from compiled code - can improve execution time and reduce footprint in the absence of C++ exception handling

- -q[no]unwind

  Asserts that the stack will not be unwound in such a way that register values must be accurately restored at call points

  Usually true in C and Fortran and allows the compiler to be more aggressive in register save/restore

  Usage of -qipa can set this option automatically in many situations

Miscellaneous Performance Options

- -qlargepage
  - Hint to the compiler that the heap and static data will be allocated from large pages at execution time (controlled by linker option -blpdata)
  - Compiler will divert large data from the stack to the heap
  - Compiler may also bias optimization of heap or static data references
- -qsmallstack
  - Tells the compiler to compact stack storage (may increase heap usage)

Directives and Pragmas

Summary of Directives and Pragmas

- OpenMP
- Legacy SMP directives and pragmas

  Most of these are superceded by OpenMP - use OpenMP
- Assertive directives (Fortran)
- Assertive pragmas (C)
- Embedded Options
- Prescriptive directives (Fortran)
- Prescriptive pragmas (C)

Overview of OpenMP (Fortran)

- Specified as directives (eg. !$OMP ...)
- PARALLEL / END PARALLEL
  - Parallel region - SPMD-style execution
  - Optional data scoping (private/shared), reductions, num_threads
- DO / END DO
  - Work sharing DO - share execution of iterations among threads
  - Optional scheduling specification (STATIC, GUIDED, etc)
- SECTIONS / SECTION / END SECTIONS
  - Share execution of a fixed number of code blocks among threads
- WORKSHARE / END WORKSHARE
  - Share execution of array assignments, WHERE or FORALL

Overview of OpenMP (continued)

- SINGLE, MASTER

  Execute block with a single thread
- CRITICAL, ATOMIC, ORDERED

  Mutual exclusion
- FLUSH, BARRIER

  Low level synchronization
- THREADPRIVATE

  Thread local copies of externals
- Runtime

  get/set num threads, low level synchronization, nested parallelism

Assertive Directives (Fortran)

- ASSERT (ITERCNT(n) | [NO]DEPS)

  Same as options of the same name but applicable to a single loop - much more useful

- INDEPENDENT

  Asserts that the following loop has no loop carried dependences - enables locality and parallel transformations

- CNCALL

  Asserts that the calls in the following loop do not cause loop carried dependences

- PERMUTATION (names)

  Asserts that elements of the named arrays take on distinct values on each iteration of the following loop

  Useful for gather/scatter codes

- EXPECTED_VALUE (param, value)

  New!

  to specify a value that a parameter passed in a function call is most likely to take at run time. The compiler can use this information to perform certain optimizations, such as function cloning and inlining. (XL Fortran V11.)

Assertive Pragmas (C)

- isolated_call (function_list)

  asserts that calls to the named functions do not have side effects
- disjoint (variable_list)

  asserts that none of the named variables (or pointer dereferences) share overlapping areas of storage
- independent_loop

  equivalent to INDEPENDENT
- independent_calls

  equivalent to CNCALL

Assertive Pragmas (C)

- permutation

  equivalent to PERMUTATION

- iterations

  equivalent to ASSERT(ITERCNT)

- execution_frequency (very_low)

  asserts that the control path containing the pragma will be infrequently executed

- leaves (function_list)

  asserts that calls to the named functions will not return (eg. exit)

Prescriptive Directives (Fortran)

- PREFETCH

  PREFETCH_BY_LOAD (variable_list): issue dummy loads to cause the given variables to be prefetched into cache - useful on Power machines or to activate Power 3 hardware prefetch

  PREFETCH_FOR_LOAD (variable_list): issue a dcbt instruction for each of the given variables.

  PREFETCH_FOR_STORE (variable_list): issue a dcbtst instruction for each of the given variables.

- CACHE_ZERO

  Inserts a dcbz (data cache block zero) instruction with the given address

  Useful when storing to contiguous storage (avoids the L2 store miss entirely)

Prescriptive Directives (Fortran)

- UNROLL

  Specified as [NO]UNROLL [(n)]

  Used to activate/deactivate compiler unrolling for the following loop.

  Can be used to give a specific unroll factor.

  Works for all loops (not just innermost).
- PREFETCH_BY_STREAM_FORWARD / BACKWARD

  Emit a dcbt encoded as a hardware stream startup

  Use LIGHT_SYNC after a block of these
- LIGHT_SYNC

  Emit a lwsync instruction
- NOSIMD

  Specifies that the following loop should not be SIMD-vectorized (PPC970)
- NOVECTOR

  Specifies that the following loop should not be vectorized

Prescriptive Directives (Fortran)

- SUBSCRIPTORDER

  Reorder the dimensions of an array
- COLLAPSE

  Reduce an entire array dimension to 1 element
- DO SERIAL

  Specify that the following loop must not be parallelized
- SNAPSHOT

  Set a legal breakpoint location with variable visibility
- UNROLL_AND_FUSE

  Used to activate unroll-and-jam for outer loops
- STREAM_UNROLL

  Used to activate innerloop unrolling for streams

Prescriptive Directives

- BLOCK_LOOP directive – Stripmining example

```
 tilesize=compute_tile_size(M)
!IBM* BLOCK_LOOP(tilesize, myloop)
      do i=1, N
!IBM* LOOPID(myloop)
         do j=1, M
              ....
         end do
      end do
```

```
 tilesize=compute_tile_size(M)
do jj=1, M, tilesize
   do i=1, N
      do j=jj, min(jj+tilesize, M)
          ...
      end do
   end do
end do
```

Prescriptive Directives

- BLOCK_LOOP directive – Loop Interchange example

```
    do i=1,N
     do j=1,N
!IBM* BLOCK_LOOP(1, myloop1)
       do k=1, M
!IBM* LOOPID(myloop1)
        do l=1, M
         ...
        end do
       end do
     end do
   end do
```

```
do i=1, N
  do j=1, N
    do ll=1, M
      do k=1, M
          l=ll
          ...
      end do
    end do
  end do
end do
```

Prefetch Directives for POWER5 (XLF V9.1 & XL C/C++ V7.0)

- Power 5 supports enhanced variants of the Data Cache Block Touch (dcbt) to provide additional control of the prefetch hardware.

- Software will be able to specify the hardware stream which should be used. the direction and length of the prefetch, and whether the data is transient.

- Support provided by new compiler directives in XLF V9.1:

  PROTECTED_UNLIMITED_STREAM_SET_GO_FORWARD(prefetch_variable, stream_id)

  PROTECTED_UNLIMITED_STREAM_SET_GO_BACKWARD(prefetch_variable, stream_id)

  PROTECTED_STREAM_SET_FORWARD(prefetch_variable, stream_id)

  PROTECTED_STREAM_SET_BACKWARD(prefetch_variable, stream_id)

  PROTECTED_STREAM_COUNT(unit_count, stream_id)

  PROTECTED_STREAM_GO

  PROTECTED_STREAM_STOP(stream_id)

  PROTECTED_STREAM_STOP_ALL

  EIEIO

- Similar support provided by new intrinsics functions in XL C/C++ V7.0

Prefetch Directives for POWER5 (XLF V9.1 & XL C/C++ V7.0)

Example 1: for short streams (less than 1024 cache lines), __protected_stream_set/count/go can be used as follows:

```
  double a[N], b[N], c[N];

  /* the size of stream a is N*sizeof(double).  If the stream size is less than
1024 cache lines, then __protected_stream_set/count/go can be used */
  __protected_stream_set(FORWARD,  a[0], 1);
   __protected_stream_count(N*sizeof(double)/CacheLineSize, 1);
  __protected_stream_set(FORWARD,  b[0], 2);
   __protected_stream_count(n*sizeof(double)/CacheLineSize, 2);
   __eieio();
  __protected_stream_go();
  for (i=0; i< N; i++) {
      c[i] = a[i]*b[i];
  }
```

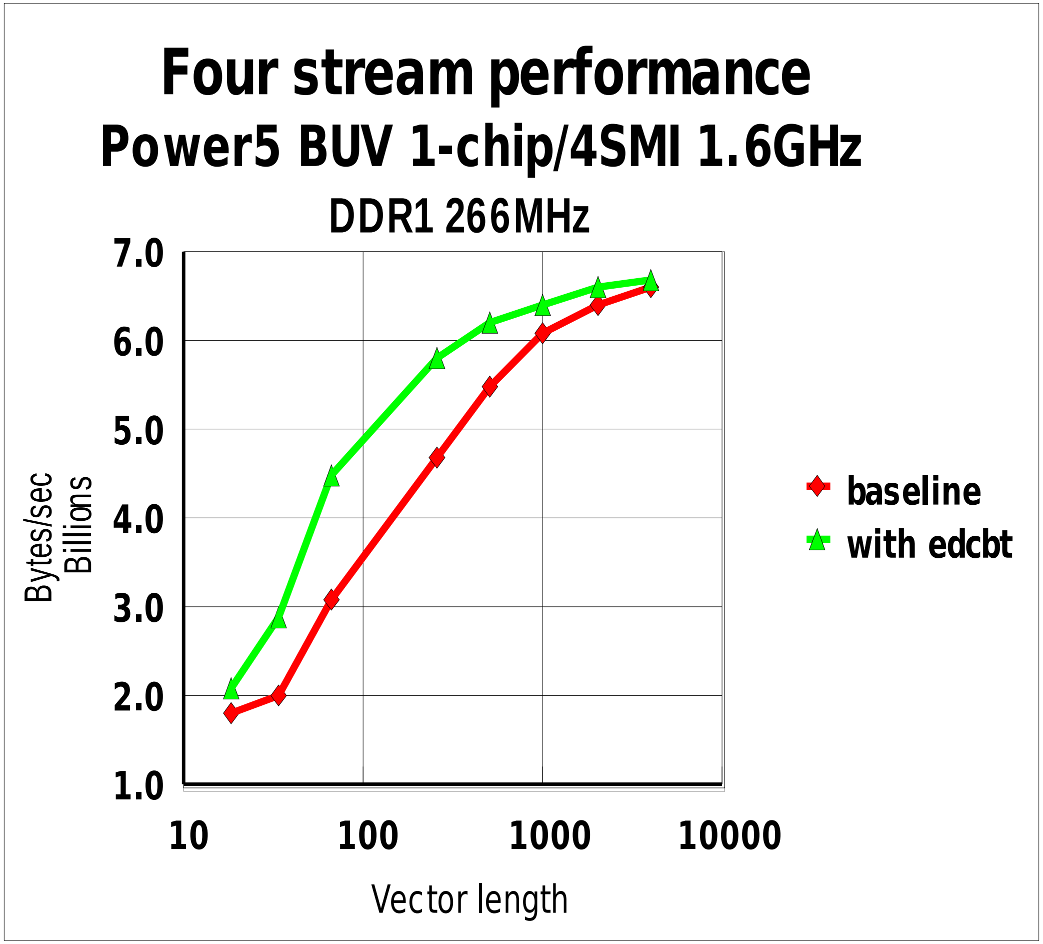
Prefetch Directives for POWER5 (XLF V9.1 & XL C/C++ V7.0)

Example 2: for long streams (equal and greater than 1024 cache lines), __protected_unlimited_stream_set_go/__protected_stream_stop can be used.

```
__protected_unlimited_stream_set_go(FORWARD, a[0], 1);
__protected_unlimited_stream_set_go(FORWARD, b[0], 2);
__protected_unlimited_stream_set_go(FORWARD, c[0], 3);
__protected_unlimited_stream_set_go(FORWARD, d[0], 4);
for (i=0; i<n; i++) {
   ...= a[i];
   ...= b[i];
   ...= c[i];
   ...= d[i];
}
__protected_stream_stop(1);
__protected_stream_stop(2);
__protected_stream_stop(3);
__protected_stream_stop(4);
```

Performance With New Prefetch Directives On POWER5

```
  do k=1,m
      lcount = nopt2
      do j=ndim2,1,-1
!!IBM  PROTECTED_STREAM_SET_FORWARD(x(1,j),0)
!!IBM  PROTECTED_STREAM_COUNT(lcount,0)
!!IBM  PROTECTED_STREAM_SET_FORWARD(a(1,j),1)
!!IBM  PROTECTED_STREAM_COUNT(lcount,1)
!!IBM  PROTECTED_STREAM_SET_FORWARD(b(1,j),2)
!!IBM  PROTECTED_STREAM_COUNT(lcount,2)
!!IBM  PROTECTED_STREAM_SET_FORWARD(c(1,j),3)
!!IBM  PROTECTED_STREAM_COUNT(lcount,3)
!!IBM  EIEIO
!!IBM  PROTECTED_STREAM_GO
       do i=1,n
        x(i,j)= x(i,j)+a(i,j)*b(i,j) + c(i,j)
       enddo
      enddo
      call dummy(x,n)
     enddo
```



Software Divide Intrinsics for POWER5 (XLF V9.1 & XL C/C++ V7.0)

- Provide alternative to hardware floating point divide instructions.
- Single precision and double precision supported :

  SWDIV(X,Y)

  where X and Y must be same type (real*4 or real*8)
  result is X / Y
  no argument value restrictions

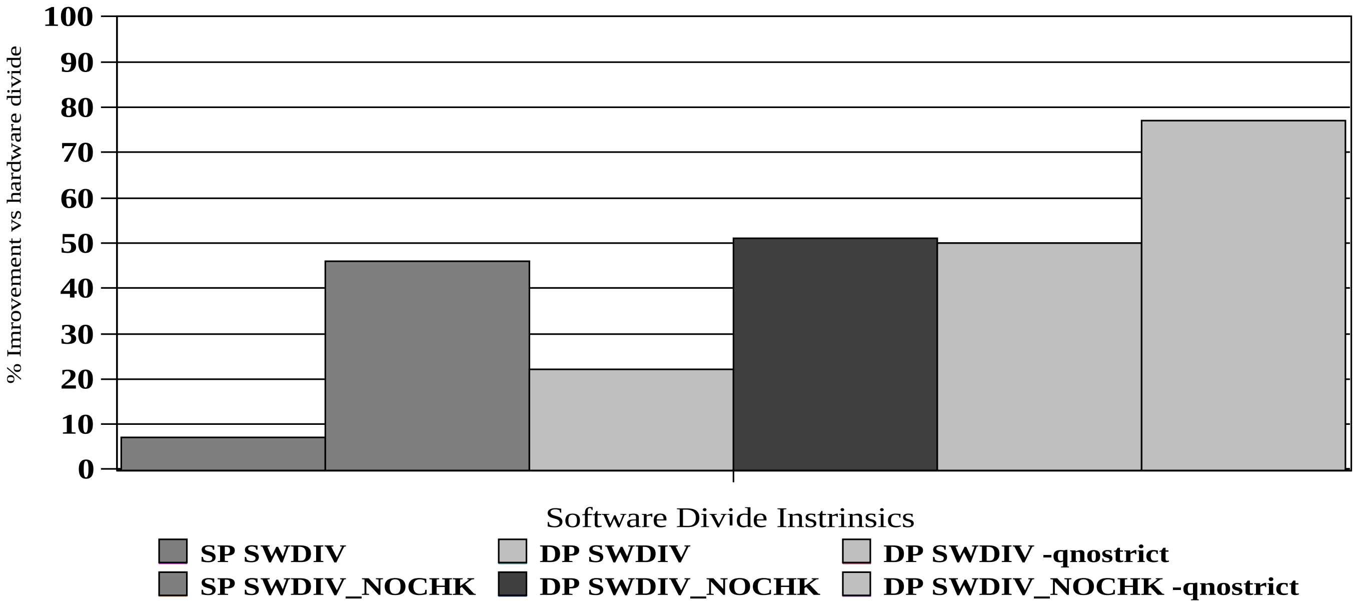
  SWDIV_NOCHK(X,Y)

  same as above except argument values cannot be infinity or denormalized. Denominator (Y) cannot be zero.

- Similar instrinsics are available in C/C++

Software Divide Intrinsics for POWER5 (XLF V9.1 & XL C/C++ V7.0)

- Must specify –qarch=pwr5 to use these intrinsics
- Compiler will automatically expand double precision divides found in loops. The profitability analysis is done as part of modulo scheduling.



All information subject to change without notice

NI Mode: An Alternate Method Of Using Faster Divides on Power5

```
subroutine sub(x,y,z,n,m)
    implicit none
    include "common.inc"
    real*8 x(*),y(*),z(*)
    include "ind_vars.inc"
    integer*4, parameter :: NI=29
    intrinsic setfsb0, setfsb1
    call setfsb1(NI)
    do j=1,m
     do i=1,n
      x(i)=z(i)/y(i)
     enddo
     call dummy(x,y,n)
    enddo
    call setfsb0(NI)
    return
    end
```

Set NI mode → call setfsb1(NI)

x(i)=z(i)/y(i) ← 27 cycles instead of 33 cycles

Reset NI mode → call setfsb0(NI)

How to Read a Compiler Listing

Compile Listing Format

Section	Description
Header	Compiler version, source file, date/time
Options	Compiler option settings Activated by -qlistopt
Source	Listing of source code and inline messages Activated by -qsource
Transformation Report	Report of transformations done by -qsmp or -qhot Activated by -qreport
Attribute and Cross-reference	Report of symbol declarations, attributes and references Activated by -qattr or -qxref
Object	Listing of generated object code and meta-data Activated by -qlist
File Table	Listing of files used in the compilation

Object Listing (-qlist)

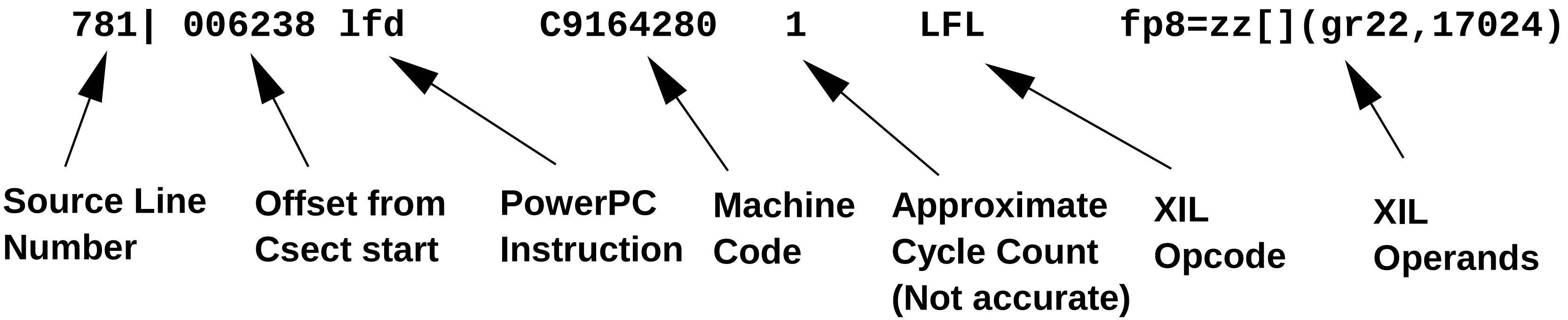
```
XL Fortran for AIX Version 08.01.0000.0002 --- kernel.f 11/05/02 22:20:25

>>>>> OPTIONS SECTION <<<<<
<snip>
>>>>> SOURCE SECTION <<<<<
** kernel   === End of Compilation 1 ===
>>>>> LOOP TRANSFORMATION SECTION <<<<<
<snip>
>>>>> OBJECT SECTION <<<<<

 GPR's set/used:   ssus ssss ssss ssss  ssss ssss ssss ssss
 FPR's set/used:   ssss ssss ssss ssss  ssss ssss ssss ssss
 CCR's set/used:   sss- ssss

     | 000000                           PDEF     kernel
    0|                                  PROC     .tk,gr3
    0| 000000 mfspr    7C0802A6   1     LFLR     gr0=lr
    0| 000004 mfcr     7D800026   1     LFCR     gr12=cr[24],2
    0| 000008 bl       4BFFFFF9   0     CALLNR   _savef14,gr1,fp14-fp31,lr"
<snip>
  830| 006B28 b        4BFF94D8   0     CALLF    _restf14
     |               Tag Table
     | 006B2C        00000000 00012223 92130000 00006B2C 1F
     |               Instruction count         6859
     |               Straight-line exec time   8381
     |               Constant Area
     | 000000        4B45524E 454C2020 00000000 49424D20 3F847AE1 47AE147B
     | 000018        40000000 3F800000 41DFFFFF FFC00000 59800004 49424D20
<snip>
```

Object Listing Line



```
781| 006238 lfd      C9164280   1      LFL      fp8=zz[](gr22,17024)
```

Source Line Number

Offset from Csect start

PowerPC Instruction

Machine Code

Approximate Cycle Count (Not accurate)

XIL Opcode

XIL Operands

XIL Cheat Sheet

LFL	Load Float Long	BT	Branch if true
STFL	Store Float Long	BF	Branch if false
L4A	Load Word Algebraic	BCT	Branch on count non-zero
ST4A	Store Word	BCF	Branch on count zero
L1Z	Load Byte and Zero	LCTR	Load Count Register
LFS	Load Float Short	CALL	Procedure Call
STFS	Store Float Short	CALLN	Procedure Call, no side effects
FMA	Long Float Multiply-Add	STFDU	Store Float Double with Update
FMS	Long Float Multiply-Subtract	CL.###:	Compiler generated label
FMAS	Single Float Multiply-Add		
AI	Add Immediate		
C4	Compare Word		
CFL	Compare Float Long		
CFS	Compare Float Short		
SRL4	Shift Right Logical		
RN4_R	Rotate and mask, with record		

But how do I find the loop?

- Look for a line number corresponding to an array store or call.

  vi: /123|/
- Look forward for the BCT.

  vi: /BCT /
- Then look backward for target label.

  vi: ?CL.1234
- The compiler normally keeps the loop contiguous so all the code between the label and the BCT is the loop. It might be useful to slice that part out.
- Loops are often unrolled and pipelined.
- Unroll "residue" usually precedes main loop. It will also be a BCT loop.
- Pipelining introduces "prologue" and "epilogue" code before and after main loop. Look for the BCF instruction to find the end of a prologue.
- Need to browse around sometimes to find the "real" loop.
- In complex cases (eg. outer unrolling, fusion, etc), it is best to get an annotated tprof listing and go to the code offsets where most of the ticks land.

Transformation Report (-qreport)

```
258|           IF ((n > 0)) THEN
                 @CSE10 = MOD(int(n), 2)
                 IF ((@CSE10 > 0)) THEN
259|               q = (z(1) * x(1))
                   IF (.NOT.(int(n) >= 2)) GOTO lab_672
                 ENDIF
258|             @PSRV0 =  0.0000000000000000E+000
                 @CIV28 = int(0)
                 @ICM.n0 = n
                 @ICM.q1 = q
                 @CSE11 = ((int(@ICM.n0) - @CSE10) + 4294967295)
                 @ICM5 = @CSE11
                 @ICM6 = (@CSE11 / 2 + 1)
    Id=3         DO @CIV28 = @CIV28, @ICM6-1
259|               @ICM.q1 = (@ICM.q1 + z((int(MOD(int(@ICM.n0), 2)) + (&
             &       @CIV28 * 2 + 1))) * x((int(MOD(int(@ICM.n0), 2)) + (&
             &       @CIV28 * 2 + 1))))
                   @PSRV0 = (@PSRV0 + z((int(MOD(int(@ICM.n0), 2)) + (@CIV28 &
             &       * 2 + 2))) * x((int(MOD(int(@ICM.n0), 2)) + (@CIV28 * 2 + &
             &       2))))
                 ENDDO
                 q = @ICM.q1
                 q = (q + @PSRV0)
                 lab_672
               ENDIF
```

```
   DO 3 K=1,N
3  Q = Q + Z(K)*X(K)
```

Transformation Report (-qreport)

```
Source        Source        Loop Id       Action / Information
File          Line
----------    ----------    ----------    ----------------------------------------------------------
         0           258             3    Inner loop has been unrolled 2 time(s).
         0           275             5    Inner loop has been unrolled 4 time(s).
         0           337            10    Loop interchanging applied to loop nest.
         0           460            17    The loops on lines 460, 469, and 478 have been fused.
         0           661                  Vectorization applied to statement.
         0           663                  Vectorization applied to statement.
         0           679                  Loop has been completely unrolled because its
                                          iteration count is less than 32.
         0           737            69    Loop interchanging applied to loop nest.
         0           737            69    Outer loop has been unrolled 4 time(s).
         0           738                  Loop has been completely unrolled because its
                                          iteration count is less than 32.
```

The MASS Library

MASS Enhancements

- Mathematical Acceleration SubSystem is a library of highly tuned, machine specific, mathematical functions available for download from IBM

    Contains both scalar and vector versions of many (mostly trig.) functions

    Trades off very limited accuracy for greater speed

    The compiler tries to automatically vectorize scalar math functions and generate calls to the MASS vector routines in libxlopt

New!

    Failing that, it tries to inline the scalar MASS routines (new for XLF V11.1)

    Failing that, it generates calls to the scalar routines instead of those in libm

MASS example

```
subroutine mass_example (a,b)
   real     a(100), b(100)
   integer           i

   do i = 1, 100
      a(i) = sin(b(i))
   enddo;
end subroutine mass_example
```

-O3 –qhot –qarch=pwr6

```
SUBROUTINE mass_example (a, b)
  @NumElements0 = int(100)
    CALL __vssin_P6 (a, b, &@NumElements0)
  RETURN
END SUBROUTINE mass_example
```

Aliasing prevents vectorization:

```
void c_example(float *a, float *b)
{
    for (int i=0; i < 100; i++)
    {
       a[i] = sin(b[i]);
       b[i] = (float) i;
    }
}
```

```
void c_example(float *a, float *b)
{
    @CIV0 = 0;
    do {
       a[@CIV0] = __xl_sin(b[@CIV0]);
       b[@CIV0] = (float) @CIV0;
       @CIV0 = @CIV0 + 1;
    } while ((unsigned) @CIV0 < 100u);
    return;
}
```

What functions are in the MASS Library?

- MASS Scalar Library (libmass.a)

  Double precision only

  exp, log, sin, cos, sincos, cosisin, tan, atan2, pow

- MASS Vector Library (libmassv.a)

  Single and double precision

  rec, div, sqrt, rsqrt, exp, log, sin, cos, sincos, cosisin, tan, atan2, dnint, dint

- POWER4 MASS Vector Library (libmassvp4.a)

  Single and double precision

  rec, div, sqrt, rsqrt, exp, log, tan, acos, asin (MASS 3.2)

New!

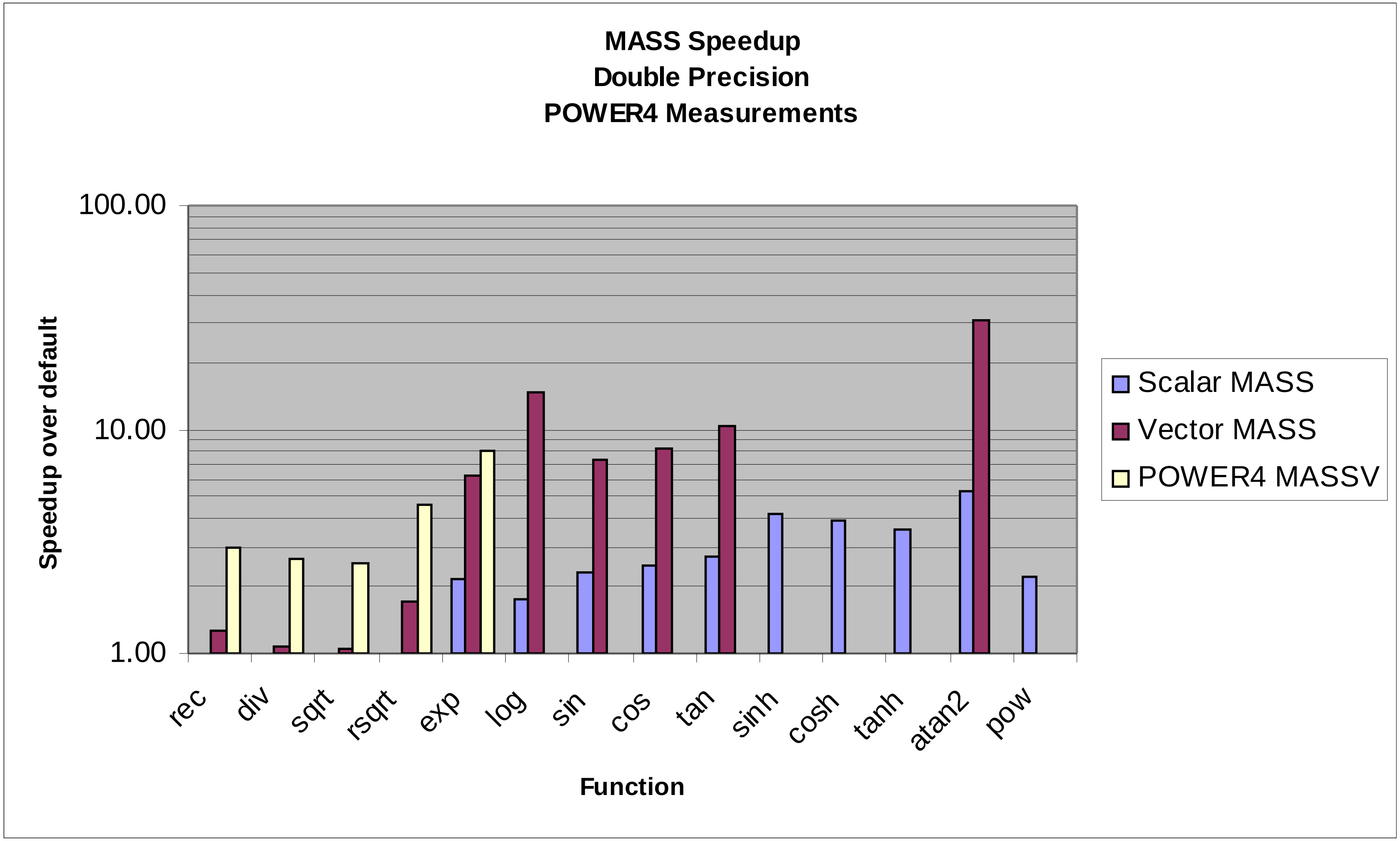
- New Power6 tuned library with 60+ routines (libmassvp6.a)

  single and double precision versions of:

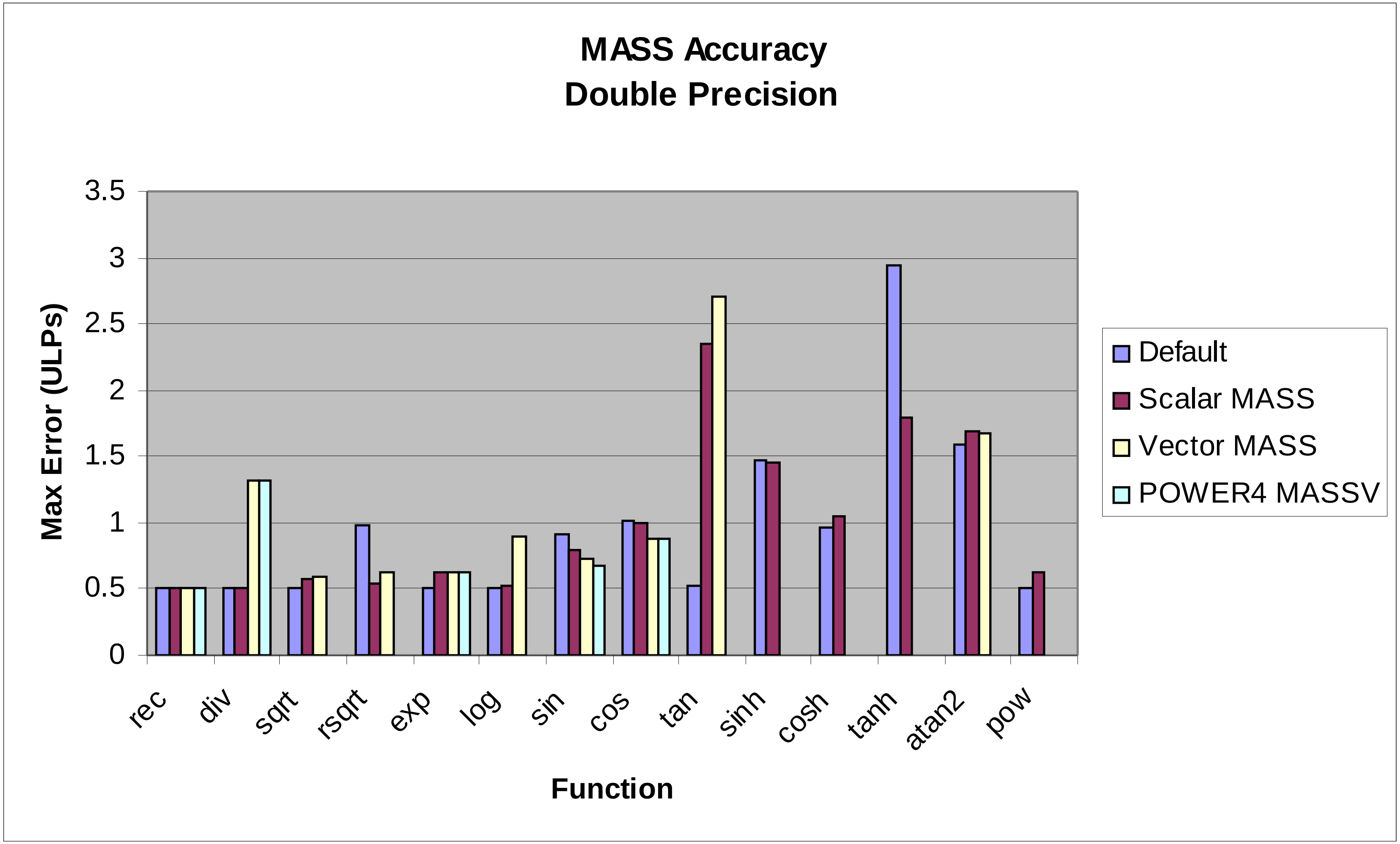
  sin(), cos(), log(), exp(), acos(), sincos() etc..

- More info: http://www.ibm.com/software/awdtools/mass/

MASS Function Performance (Double precision)



MASS Function Accuracy (Double precision)



Shared Memory Parallelism

Compiling Code for SMP

- Use the *reentrant* compiler invocations ending in “_r” such as xlf90_r or xlC_r
- The –qsmp option is used to activate parallel code generation and optimization
- Specify –qsmp=omp to compile OpenMP code

  -qsmp=omp:noopt will disable most optimizations to allow for full debugging of OpenMP programs

  Controls are also available to change default scheduling, allow nested parallelism or safe recursive locking
- Specify –qsmp=auto to request automatic loop parallelization
- Detect a thread's stack going beyond its limit (XL Fortran V11.1 and XL C/C++ V9.0). Implemented with –qsmp=stackcheck.

New!

OpenMP vs. Automatic Parallelization

- OpenMP and automatic parallelization are currently not allowed to be done together
- OpenMP is recommended for those who are able to expend the effort of annotating their code for parallelism
  - More flexible than automatic parallelization
  - Portable
- Automatic parallelization is recommended as a means of doing some parallelization without code changes
- Automatic parallelization along with -qreport can be helpful for identifiying parallel loop opportunities for an OpenMP programmer

New!

- -qsmp=threshold=n to specify the amount of work required in a loop before the compiler considers it for automatic parallelization

Running SMP Code

- Ensure the stack size is large enough
  - Use of array language may increase stack size unexpectedly
  - export XLSMPOPTS=STACK=nn
  - Consider using the -qsmallstack option to allocate less on the stack
- Set SPINS and YIELDS to 0 if using the machine or node in a dedicated manner
  - export XLSMPOPTS=SPINS=0:YIELDS=0
- By default, the runtime will use all available processors.
  - Do not set the PARTHDS or OMP_NUM_THREADS variables unless you wish to use fewer than the number of available processors
- One can bind SMP threads to processor threads using startproc and stride
  - For example, export XLSMPOPTS=startproc=0:stride=2
  - Binds SMP thread 0 to processor thread 0 and SMP thread 1 to processor thread 2…
- The number of SMP threads used by MATMUL and RANDOM_NUMBER can be controlled.
  - For example, export XLSMPOPTS=intrinthds=2
  - Will allow only 2 SMP threads to be used by these intrinsics

Performance Programming

What can my compiler do for me?

- **Code generation and register allocation**
  - Redundancy and store elimination
  - Register allocation
- **Instruction scheduling**
  - Global and local reordering
  - Loop pipelining
- **Program restructuring**
  - Inlining and specialization
  - Partitioning
- **Loop restructuring**
  - Fusion and fission
  - Unrolling
  - Vectorization and Parallelization

Redundancy Elimination

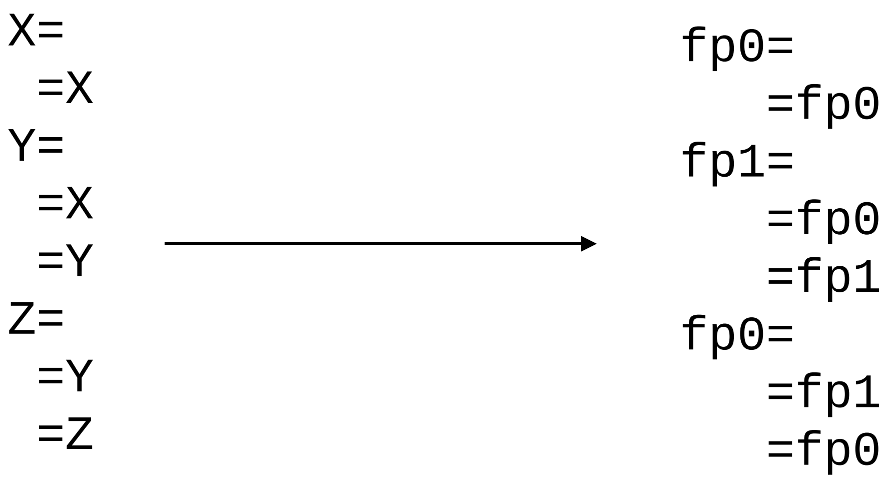
```
for (i=0;i<n;i++)
  for (j=0;j<m;j++)
    a[i][j] = b[j][i];
```

Key points:

- Relies on good alias information
- Depends largely on original expression ordering

```
pa_b = a;
pb_b = b;
for (i=0;i<n;i++) {
  pa = pa_b + a_colsz;
  pb = pb_b + i*datasz;
  for (j=0;j<m;j++) {
    *pa++ = *pb;
    pb = pb + b_colsz;
  }
}
```

Register Allocation



```
X=
  =X
Y=
  =X
  =Y
Z=
  =Y
  =Z
```

```
fp0=
   =fp0
fp1=
   =fp0
   =fp1
fp0=
   =fp1
   =fp0
```

Key points:
•Relies on good alias information
•Speed and quality of allocation degrade with size of function

Instruction Scheduling

```
DO I=1,N,2
  A(I)=A(I)+P*B(I)+Q*C(I)
  A(I+1)=A(I+1)+P*B(I+1)+Q*C(I+1)
END DO
```

Key points:
- Relies on good alias information
- Limited in scope
- Less effective across branches

```
lfd     fp0,A(I)
lfd     fp1,B(I)
fma     fp0,fp0,fp30,fp1
lfd     fp2,C(I)
fma     fp0,fp0,fp31,fp2
stfd    fp0,A(I)
lfd     fp3,A(I+1)
lfd     fp4,B(I+1)
fma     fp3,fp3,fp30,fp4
lfd     fp5,C(I+1)
fma     fp3,fp3,fp31,fp5
stfd    fp3,A(I+1)
```

```
lfd     fp0,A(I)
lfd     fp1,B(I)
lfd     fp3,A(I+1)
lfd     fp4,B(I+1)
fma     fp0,fp0,fp30,fp1
fma     fp3,fp3,fp30,fp4
lfd     fp2=C(I)
lfd     fp5=C(I+1)
fma     fp0,fp0,fp31,fp2
fma     fp3,fp3,fp31,fp5
stfd    fp0,A(I)
stfd    fp3,A(I+1)
```

Loop Pipelining

```
DO I=1,N,4
  A(I)=A(I)+P*B(I)+Q*C(I)
  A(I+1)=A(I+1)+P*B(I+1)+Q*C(I+1)
  A(I+2)=A(I+2)+P*B(I+2)+Q*C(I+2)
  A(I+3)=A(I+3)+P*B(I+3)+Q*C(I+3)
END DO
```

```
A1=A(1) A2=A(2) A3=A(3) A3=A(4)
B1=B(1) B2=B(2) B3=B(3) B4=B(4)
C1=C(1) C2=C(2) C3=C(3) C4=C(4)
DO I=4,N,4
  A(I-4)=A1+P*B1+Q*C1
  A1=A(I) B1=B(I) C1=C(I)
  A(I-3)=A2+P*B2+Q*C2
  A2=A(I+1) B2=B(I+1) C2=C(I+1)
  A(I-2)=A3+P*B3+Q*C3
  A3=A(I+2) B3=B(I+2) C3=C(I+2)
  A(I-1)=A4+P*B4+Q*C4
  A4=A(I+3) B4=B(I+3) C4=C(I+3)
END DO
A(I)=A1+P*B1+Q*C1
A(I+1)=A2+P*B2+Q*C2
A(I+2)=A3+P*B3+Q*C3
A(I+3)=A4+P*B4+Q*C4
```

Key points:

- Relies on good alias information
- Limited in scope to small loops with no branches
- Effectiveness depends on unrolling to sufficient depth

Function Inlining

```
SUBROUTINE C(X,Y)
CALL B(Y*2,X)
END
SUBROUTINE B(P,Q)
CALL A(Q,P*3)
END
SUBROUTINE A(R,S)
R = R * S
END
PROGRAM FOO
CALL C(F,4.0)
END
```

```
F=F*24.0
```

Key points:

- Caller and callee size limits important parameters
- Calls considered in order of expected frequency (better with PDF)
- Inlining can be harmful due to growing code beyond other optimizations’ effective limits

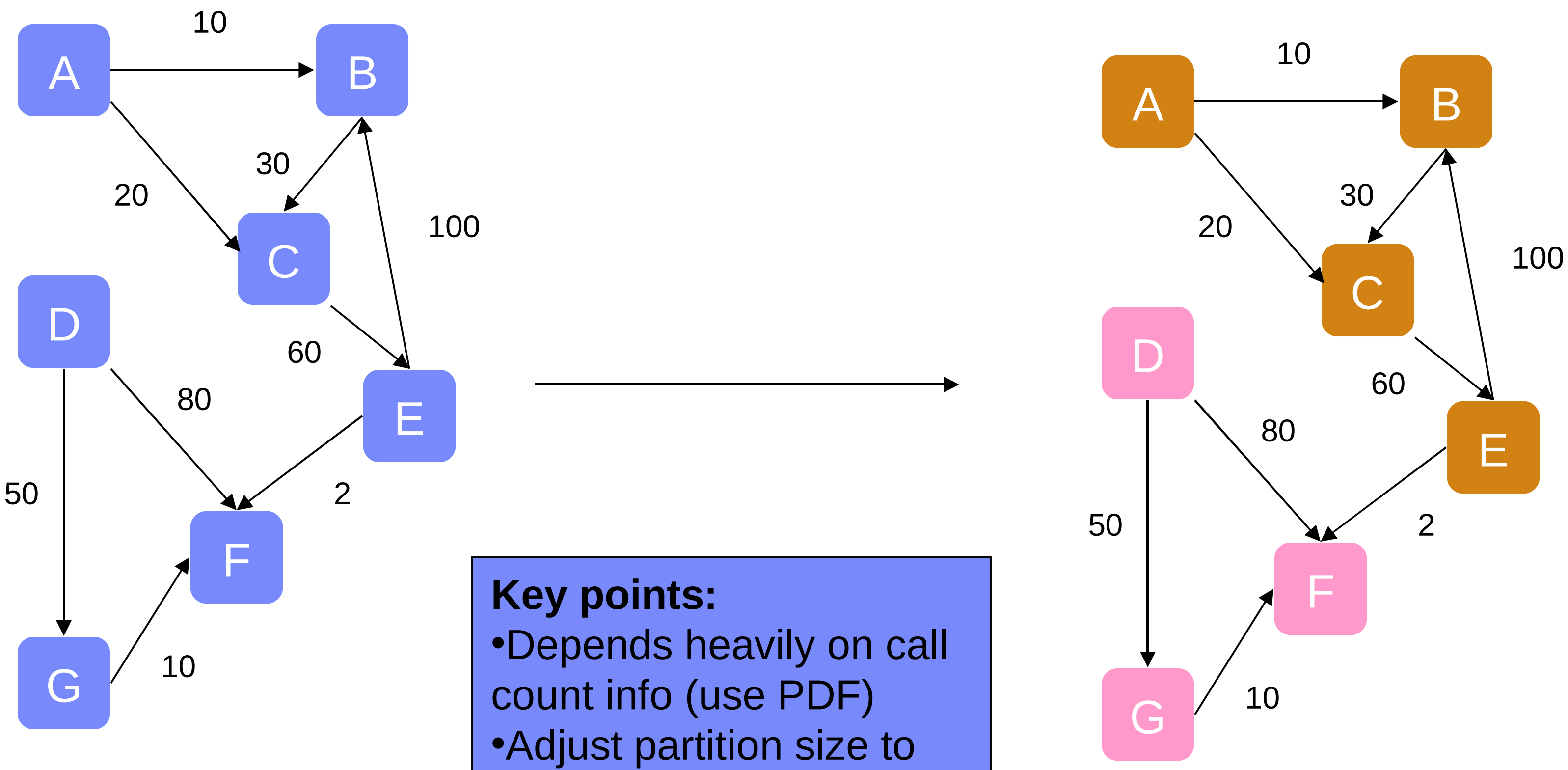
Function Specialization

```
int f(x) {
  z=x*4.0
  return fsqrt(z);}
print(f(x),z);
y=f(x/7.0)
print(z);
... never use y ...
```

Key points:
•Active only with –O5
•Not effective at recognizing aliasing specializations

```
double f(x) {
  z=x*4.0
  return fsqrt(z);}
double f0(x) {
  z=x*4.0;
  return 0.0;}
print(f(x),z);
y=f0(x/7.0)
print(z);
```

Program Partitioning



Key points:
•Depends heavily on call count info (use PDF)
•Adjust partition size to trade effectiveness for increased link time

Loop Fusion

```
DO I=1,N
  A(I)=F(B(I))
END DO
Q =
DO J=2,N
  C(I)=A(I-1)+ Q*B(I)
END DO
```

Key points:
- Limited by dependence and alias information
- Exploit reuse
- Requires *countable* loops (more on this later)

```
Q =
A(1)=F(B(1))
DO I=2,N
  A(I)=F(B(I))
  C(I)=A(I-1)+Q*B(I)
END DO
```

Loop Fission

```
DO I=1,N
  A(I)=A(I)+P1*B1(I)+P2*B2(I)+P3*B3(I)+P4*B4(I)+
            P5*B5(I)+P6*B6(I)+P7*B7(I)+P8*B8(I)+
            P9*B9(I)
END DO
```

Key points:

- Limited by dependence and alias information
- Requires *countable* loops
- Reduce number of streams

```
DO I=1,N
  A(I)=A(I)+P1*B1(I)+P2*B2(I)+P3*B3(I)+P4*B4(I)
END DO
DO I=1,N
  A(I)=A(I)+P5*B5(I)+P6*B6(I)+P7*B7(I)+P8*B8(I)+
            P9*B9(I)
END DO
```

Loop Unrolling

```
DO I=1,N
  S = S + A(I)*B(I)
END DO
```

```
DO I=1,N,4
  S1 = S1 + A(I)*B(I)
  S2 = S2 + A(I+1)*B(I+1)
  S3 = S3 + A(I+2)*B(I+2)
  S4 = S4 + A(I+3)*B(I+3)
END DO
S = S1+S2+S3+S4
```

Key points:

- Unroll factor selection is a balance between scheduling and register allocation
- Requires *countable* loops

Loop Vectorization

```
DO I=1,N
  S = B(I) / SQRT(C(I))
  A(I) = LOG(S)*C(I)
END DO
```

```
CALL VRSQRT(A,C,N)
DO I = 1,N
  A(I) = B(I)*A(I)
END DO
CALL VLOG(A,A,N)
DO I = 1,N
  A(I) = A(I)*C(I)
END DO
```

Key points:

- Vectorization requires fission
- Vectorization is a balance between vector speedup and memory locality

Loop Parallelization

```
DO I=1,N
  DO J=2,M-1
    A(I,J) = A(I,J-1)*SQRT(A(I,J+1))
  END DO
END DO
```

Key points:

•Limited by dependence and alias information

•Loop selection imperfect due to lack of knowledge about bounds

```
CALL _xlsmpDo (FOO@OL@1,1,N)
...
SUBROUTINE FOO@OL@1(LB,UB)
DO I=LB,UB
  DO J=2,M-1
    A(I,J) = A(I,J-1)*SQRT(A(I,J+1))
  END DO
END DO
END
```

What can I do for my compiler?

- Simplify memory accesses
  - Use of externals and pointers
  - Array indexing
- Simplify loop structures
  - Countable loops
  - Perfect nesting
  - Branches in loops
- Simplify program structure
  - When to inline
  - When to use function pointers

Use of externals

- The optimizer scope is often a single function
  - External variables *might* have their address taken in some other function and so might be used or modified by certain pointer dereferences
  - External variables *might* be used of modified by a call to some other function
- Use *scalar replacement* to manually allocate external variables to registers (ie. automatic variables that do not have their address taken).
- Increase optimizer scope
  - Use modules in Fortran
  - Group related functions in the same file and make as many *static* as possible in C/C++

Array indexing

- Many optimizations attempt to *disambiguate* array references by an analysis of index expressions
- Simplify index calculations
  - Keep calculations as a linear function of the loop variables where possible
  - Avoid unnecessary use of shift operators, modulus or divide
  - Use language facilities for indexing where possible (rather than manually indexing off a pointer for example)
- Simplfy loop bound expressions
  - Store bounds to temporaries and use the temporaries in the loop
  - Avoid unnecessary use of shift operators, modulus or divide
  - Avoid unnecessary use of MAX/MIN
- Use a single name for each array to avoid *aliasing* ambiguities
- Consider using the INDEPENDENT directive

Use of pointers

- Pointers can be confusing for the compiler because dereferencing them may imply a use or definition of any other variable which has had its address taken
- Obey the strictest language rules available as a matter of practice
  - Type-based language rules for pointers are very useful in refining the alias sets of dereferences
  - Use TARGET only when necessary in Fortran
  - Enhances portability
- Take the address of a variable <u>only when necessary</u>
- Reference parameters
  - Parameters passed by address in C, C++ or Fortran effectively have their address taken
  - Exploit Fortran parameter aliasing rules by encapsulating operations on disjoint sections of the same array

Countable loops

- Most loop optimizations in the compiler work only on *countable* loops
  - A countable loop has a single entry point, a single exit point and an iteration count which can be determined before the loop begins
- Use counted DO loop or canonical *for* statement for as many loops as possible – these are usually countable
- Use *register* index variables and bounds
  - Simulate a register using an automatic whose address is not taken
  - Avoids aliasing ambiguity
- Branches in/out of loops
  - Never branch into a loop
  - Isolate exit conditions into their own *search* loop where possible

Perfect nesting

- Many loop optimizations, including outer unroll-and-jam and interchange require *perfect loop nesting*
  - A loop is perfectly nested in another if there is no other code executed in the containing loop
- Split loops where possible to create perfect nests
- Definitions of *register* variables are allowed so no need to avoid them

Branches in Loops

- Branches in loops decrease the effectiveness of loop fission, vectorization, unrolling and pipelining

```
DO I=1,N
  IF (C>0.0) THEN
    A(I)=A(I)+B(I)
  ELSE
    A(I)=A(I)+C*B(I)
  END IF
END DO
```

↓

```
IF (C>0.0) THEN
  DO I=1,N
    A(I)=A(I)+B(I)
  END DO
ELSE
  DO I=1,N
    A(I)=A(I)+C*B(I)
  END DO
END IF
```

```
DO I=1,N
  IF (I<5) THEN
    A(I)=A(I)+B(I)
  ELSE
    A(I)=A(I)-B(I)
  END IF
END DO
```

↓

```
DO I=1,4
  A(I)=A(I)+B(I)
END DO
DO I=5,N
  A(I)=A(I)-B(I)
END DO
```

```
DO I=1,N
  IF (C(I)>0.0) THEN
    A(I)=A(I)/B(I)
END DO
```

↓

```
NT=1
DO I=1,N
  IF (C(I)>0.0) THEN
    IX(NT) = I
    NT=NT+1
  END IF
END DO
DO II=1,NT-1
  IT=IX(II)
  A(IT)=A(IT)/B(IT)
END DO
```

When to inline

- Inlining is a difficult decision for the compiler to make on its own
- C/C++ have *inline* keywords to help identify functions which should be inlined
- -qipa has *inline* and *noinline* options (which can be used with regular expressions and a side file) to guide inlining decisions
- Profile directed feedback is very effective at identifying the best inlining candidates
- Manually inline a function only when you absolutely need to do it for performance

Function pointer use

- Function pointers in C or C++ are very confusing to the compiler because the destination of a call through pointer is not known in general
- C++ virtual methods are equivalent to function pointers in this respect
- For a small number of pointer targets, consider using a switch with direct calls in the cases instead
- If there are a large number of possible targets but a small number of probable targets, create a switch with the common values and include the pointer call in the *default* clause
- If most pointer targets are in the same shared object or executable as the caller, consider creating local “stubs” for the targets which are outside

VMX Exploitation

VMX Exploitation

- **User directed**

  Vector data types and routines available for C, C++ and Fortran

  Programmer manually re-writes program, carefully adhering to the alignment constraints

- **Automatic SIMD Vectorization (SIMDization)**

  The compiler automatically identifies parallel operations in the scalar code and generates SIMD versions of them.

  The compiler performs all analysis and transformations necessary

  to fulfill alignment constraints.

  Programmer assistance may improve generated code

Required Options for VMX Code Generation

- For programs with VMX intrinsics:

  C/C++: -qaltivec -qarch=pwr6

  Fortran: -qarch=pwr6

- Automatic SIMD vectorization:

  Optimization level -O3 -qhot or higher and -qarch=pwr6

- -q[no]enablevmx - Compiler is allowed to generate VMX instructions

  AIX defaults to -qnoenablevmx (must be explicitly turned on by user)

  Linux defaults to -qenablevmx

User Directed VMX

- Data types:

  C/C++: vector float, vector int, vector unsigned char

  Fortran: vector(real(4)), vector (integer), vector(unsigned(1))

- VMX intrinsics

  vec_add(), vec_sub(), vec_ld(), vec_st(), etc.

  The Fortran VMX intrinsic names are the same as those of C/C++

- Symbolic debug (gdb, dbx) support at no-opt.

- Fully optimized at -O2 and above with suite of classical optimizations such as dead code removal, loop invariant code motion, software pipelining and Power6 instruction scheduling

Example: Fortran VMX Intrinsics

```
subroutine xlf_madd (a,b,c,x)
    vector(real(4))     a(100), b(100), c(100)
    vector(real(4))     x(100)
    integer             i

    do i = 1, 100
      x(i) = vec_madd(a(i), b(i), c(i))
    enddo;
end subroutine xlf_madd
```

```
          CL.5:
VLQ      vr0=a[](gr4,gr7,0)
VLQ      vr1=b[](gr5,gr7,0)
VLQ      vr2=c[](gr6,gr7,0)
VMADDFP  vr0=vr0-vr2,nj
VSTQ     x[](gr3,gr7,0)=vr0
AI       gr7=gr7,16
BCT      ctr=CL.5,,100,0
```

Compile options:

```
xlf -O2  -qarch=pwr6 -qlist -c
```

Additionally, compiling on AIX requires -qenablevmx

Successful Simdization

Extract Parallelism

loop level

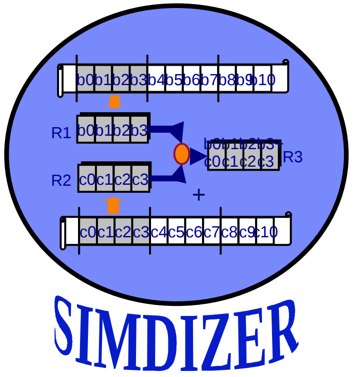
```
for (i=0; i<256; i++)
  a[i] =
```

basic-block level

```
a[i+0] =
a[i+1] =
a[i+2] =
a[i+3] =
```

entire short loop

```
for (i=0; i<8; i++)
  a[i] =
```


b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10
R1 b0 b1 b2 b3
R2 c0 c1 c2 c3
b0 b1 b2 b3
c0 c1 c2 c3
R3
+
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 c10
SIMDIZER

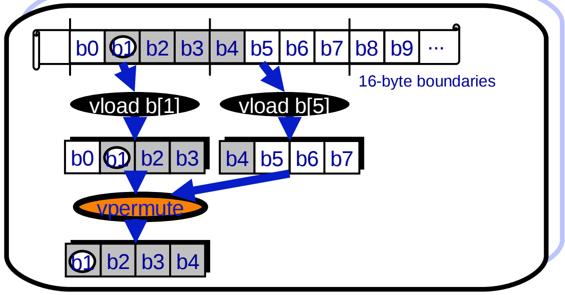
multiple targets

GENERIC

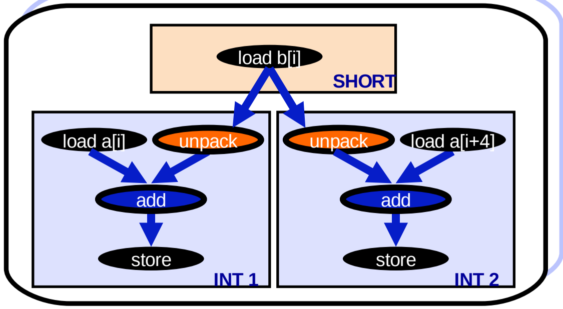
VMX SPE

Satisfy Constraints

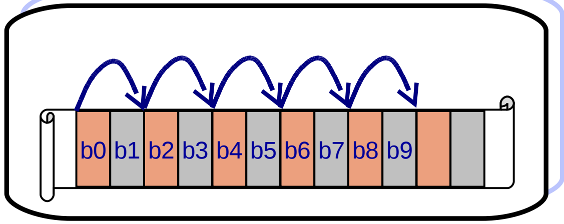
alignment constraints



data size conversion



non stride-one



Coding choices that impact simdization

- How loops are organized
  - Loop must be countable, preferably with literal trip count
  - Only innermost loops are candidates for simdization, except when nested loops have a short literal iteration count
  - Loops with control flow are harder to simdize. Compiler tries to remove control flow, but not always successful
- How data is accessed and laid out in memory
  - Data accesses should preferably be stride-one
  - Layout the data to maximize aligned accesses
  - Prefer use of arrays to pointer arithmetic
- Dependences inherent to the algorithm
  - Loops with inherent data dependences are not simdizable
  - Avoid pointers; pointer aliasing may impede transformations

Assisting the compiler to perform auto-SIMD

- Loop structure

  Inline function calls inside innermost loops

  Automatically (-O5 more aggressive, use inline pragma/directives)

- Data alignment

  Align data on 16-byte boundaries

  __attribute__((aligned(16))

  Describe pointer alignment

  _alignx(16, pointer)

  Can be placed anywhere in the code, preferably close to the loop

  Use -O5 (enables inter-procedural alignment analysis)

- Pointer aliasing

  Refine pointer aliasing #pragma disjoint(*p, *q) or restrict keyword

  Use -O5 (enables interprocedural pointer analysis)

AIX and -qvecnvol

- Compiler does not use non-volatile VMX registers on AIX
  - -qvecnvol is default on AIX
  - -qnovecnvol forces the use of non-volatile registers, default on Linux
- May cause severe performance impact
  - Can use -qnovecnvol on AIX if your code does not call legacy modules
- If legacy modules that do setjmp/longjmp/sigsetjmp/siglongjmp are linked with new VMX objects, it may produced incorrect results
- Example scenario:

```
new (VMX) module calls old (non-VMX) module
  old (non-VMX) module performs setjmp() [non-vol VMX not saved]
        calls another new (VMX) module [non-vol VMX state might be modified]
         if new module returns [ok, non-vol is restored by linkage convention]
         if new module longjmps to "old" jmpbuf [problem, non-vol VMX state not restored]
         calls old (VMX) module
            if old (VMX) module longjmps [problem, non-vol VMX state not restored]
```

Did we SIMDize the loop?

- The -qreport option produces a list of high level transformation performed by the compiler
  - Everything from unrolling, loop interchange, SIMD transformations, etc.
  - Also contains transformed “pseudo source”
- All loops considered for SIMDization are reported
  - Successful candidates are reported
  - If SIMDization was not possible, the reasons that prevented it are also provided
- Can be used to quickly identify opportunities for speedup

Example – SIMD Problems Reported

```
extern int *b, *c;

int main()
{
    for (int i=0; i<1024; ++i)
        b[i+1] = b[i+2] - c[i-1];
}
```

```
1586-535 (I) Loop (loop index 1) at d.c <line 9> was not SIMD vectorized because the aliasing-induced dependence
prevents SIMD vectorization.
1586-536 (I) Loop (loop index 1) at d.c <line 9> was not SIMD vectorized because it contains memory references  with
non-vectorizable alignment.
1586-536 (I) Loop (loop index 1) at d.c <line 11> was not SIMD vectorized because it contains memory references
((char *)b + (4)*((@CIV0 + 1))) with non-vectorizable alignment.
1586-543 (I) <SIMD info> Total number of the innermost loops considered <"1">. Total  number of the innermost loops
SIMD vectorized <"0">.
```

```
    5 |   long main()
          {
    9 |     @ICM.b0 = b;
            if (!1) goto lab_5;
            @CIV0 = 0;
            __prefetch_by_stream(1,((char *)@ICM.b0  + (0 - 128) + (4)*(@CIV0 + 2)))
            __iospace_lwsync()
   11 |     @ICM.c1 = c;
    9 |     do {   /* id=1 guarded */ /* ~4 */
              /* region = 8 */
              /* bump-normalized */
   11 |       @ICM.b0[@CIV0 + 1] = @ICM.b0[@CIV0 + 2] - @ICM.c1[@CIV0 - 1];
    9 |       @CIV0 = @CIV0 + 1;
            } while ((unsigned) @CIV0 < 1024u);    /* ~4 */
          lab_5:
            rstr = 0;
   14 |     return rstr;
          } /* function */
```

Example: Correcting SIMD Inhibitors

```
extern int * restrict b, * restrict c;


int main()
{
  /* __alignx(16, c);    Not strictly required since compiler   */
  /* __alignx(16, b);    inserts runtime alignment check       */

   for (int i=0; i<1024; ++i)
      b[i] = b[i] - c[i];
}
```

```
586-542 (I) Loop (loop index 1 with nest-level 0 and iteration count 1024) at d_good.c <line 9>
was SIMD vectorized.
1586-542 (I) Loop (loop index 2 with nest-level 0 and iteration count 1024) at d_good.c <line 9>
was SIMD vectorized.
1586-543 (I) <SIMD info> Total number of the innermost loops considered <"2">. Total number of the innermost loops
SIMD vectorized <"2">.

     7 |  long main()
          {
            @ICM.b0 = b;
            @ICM.c1 = c;
     9 |    @ICMB = (0 - 128);
            @ICM4 = (long) @ICM.c1 & 15;
            @CSE2 = (long) @ICM.b0;

             .  .  .
```

Other Examples of SIMD Messages

- Loop was not SIMD vectorized because it contains operation which is not suitable for SIMD vectorization.
- Loop was not SIMD vectorized because it contains function calls.
- Loop was not SIMD vectorized because it is not profitable to vectorize.
- Loop was not SIMD vectorized because it contains control flow.
- Loop was not SIMD vectorized because it contains unsupported vector data types
- Loop was not SIMD vectorized because the floating point operation is not vectorizable under -qstrict.
- Loop was not SIMD vectorized because it contains volatile reference

Other SIMD Tuning

- **Loop unrolling can interact with simdization**

  Manually-unrolled loops are more difficult to simdize

- **Tell compiler not to simdize a loop if not profitable**

  #pragma nosimd (right before the innermost loop)

  Useful when loop bounds are small and unknown at compile time

Programming for POWER6

Compiling for Power6

- New -qarch suboptions for Power6:

-qarch=pwr6e	- Generate all P6 instructions
-qarch=pwr6	- Generate all except for raw-mode only instructions

- Some P6 instructions are only available when the P6 is in “raw mode”

mffgpr, mftgpr:	move between float and integer registers

- Using -qarch=pwr6 will ensure that your binaries continue to run on upcoming processors, while -qarch=pwr6e may provide additional performance.

Power5 / Power6 differences (summary)

- Power6 executes instructions in order

  Helps to reach high clock rate, but more stalls
- Store Queue has to be managed to prevent load/store stalls

  Careful arrangement of stores can get the bandwidth back in some cases
- Power6 does not do store forwarding

  High cost for store and reload
- Fixed point multiplies are done in the floating point unit

  Extra cost can be mitigated by grouping them
- VMX and DFP unit available

Power5 / Power6 differences (summary)

- Power6 executes instructions in order
  - Helps to reach high clock rate, but more stalls
- Store Queue has to be managed to prevent load/store stalls
  - Careful arrangement of stores can get the bandwidth back in some cases
- Power6 does not do store forwarding
  - High cost for store and reload
- Fixed point multiplies are done in the floating point unit
  - Extra cost can be mitigated by grouping them
- VMX and DFP unit available

Balanced Tuning (-qtune=balanced)

- This is a new compiler tuning target
- We try to balance the competing optimization priorities of Power5 and Power6
  - Insert special group ending NOP when required, on P5 this acts just like a regular NOP
  - Have “loads only” and “stores only” groups when possible
  - Group fixed point multiplies together in a sequence
- This is available in a recent PTF of XLC/C++ V8.0 and XLF V10.1
- This tuning option becomes the default in XLC/C++ V9.0 and XLF V11.1

Prefetch Enhancements for P6

- Exploit the 16 streams available on Power6 (only 8 on P4/P5)
- Support new store stream prefetch

  Compiler automatically determines when prefetch insertion is profitable

  and inserts calls to prefetch stores

- Exploit both L1 and L2 touch instructions

  Compiler automatically determines if data is more likely to be needed in L1

  or L2 and inserts the prefetch required.

- Exploit prefetch depth control

  Try to fetch further ahead

  Tricky to get right, may compete with immediately needed lines in L1

Decimal Floating Point (DFP) Support

- The XLC/C++ V9 compilers and AIX 5.3 will support DFP.

  Family of floating-point types where the fraction is decimal digits instead of binary bits.

- New C/C++ data types, printf() format specifiers, etc.

  ```
  _Decimal32 (7 digits)
  _Decimal64 (16 digits)
  _Decimal128 (34 digits)
  ```

- The V9 compilers on Linux will support DFP as a technical preview.
- Power6 supports DFP in hardware

  Compiler supports DFP via hardware exploitation as well as with calls to DFP software library.

- Full IEEE / C / C++ compliance (eg, complete math library, some new IEEE features) will be provided later.

DFP Support (cont.)

- New compile option -qdfp

  Enable DFP types, literals and functions.

- -qfloat=[no]dfpemulate

  Controls using hardware instructions or software emulation on PowerPCs.

  Default is to use hardware on Power6, software on other models.

- Examples:

  ```
  xlc  foo.c  -qdfp  -qarch=pwr6      # uses hw instructions
  xlc  bar.c  -qdfp  -qarch=pwr5     # uses sw emulation
  ```