Make Your PL/I and C/C++ Code Fly With the Right Compiler Options

Visda Vokhshoori
visdav@ca.ibm.com

WHAT …

- does good application performance mean to you?

  - Fast Execution Time

  - Short Compile Time

Can you please take this poll: http://p2.gg/itp

HOW …

- to achieve good application performance:

  - Install New Hardware

  - Utilize Compiler Options

  - Code for Performance

Install New Hardware

- Can make your code run faster

- Requires NO
  - Recompilation
  - Relinking
  - Migration to new release

- BUT, are you taking full advantage of all the new features from the new hardware?
  - i.e. the full ROI on the new piece of hardware

Utilize Compiler Options

- Allows the compiler to exploit the hardware:

  - ARCH

  - HGPR

  - FLOAT(AFP)

- Balance between compile-time vs. execution-time:

  - OPT(2)

  - OPT(3)

  - HOT [C/C++]

  - IPA [C/C++]

  - PDF

Utilize Compiler Options (cont'd)

- Provide the details about the source or environment:

  - C/C++:

    - ANSIALIAS

    - IGNERRNO

    - LIBANSI

    - NOTHREADED

    - NOSTRICT

    - STRICT_INDUCTION

    - XPLINK

  - PL/I:

    - REDUCE

    - RESEXP

- Controls load module size:

  - COMPACT [C/C++]

  - INLINE [C/C++]

  - DEFAULT(INLINE) [PL/I]

  - UNROLL

Code for Performance

- Writing good code

- Make use of built-in functions

- Make use of #pragmas [C/C++]

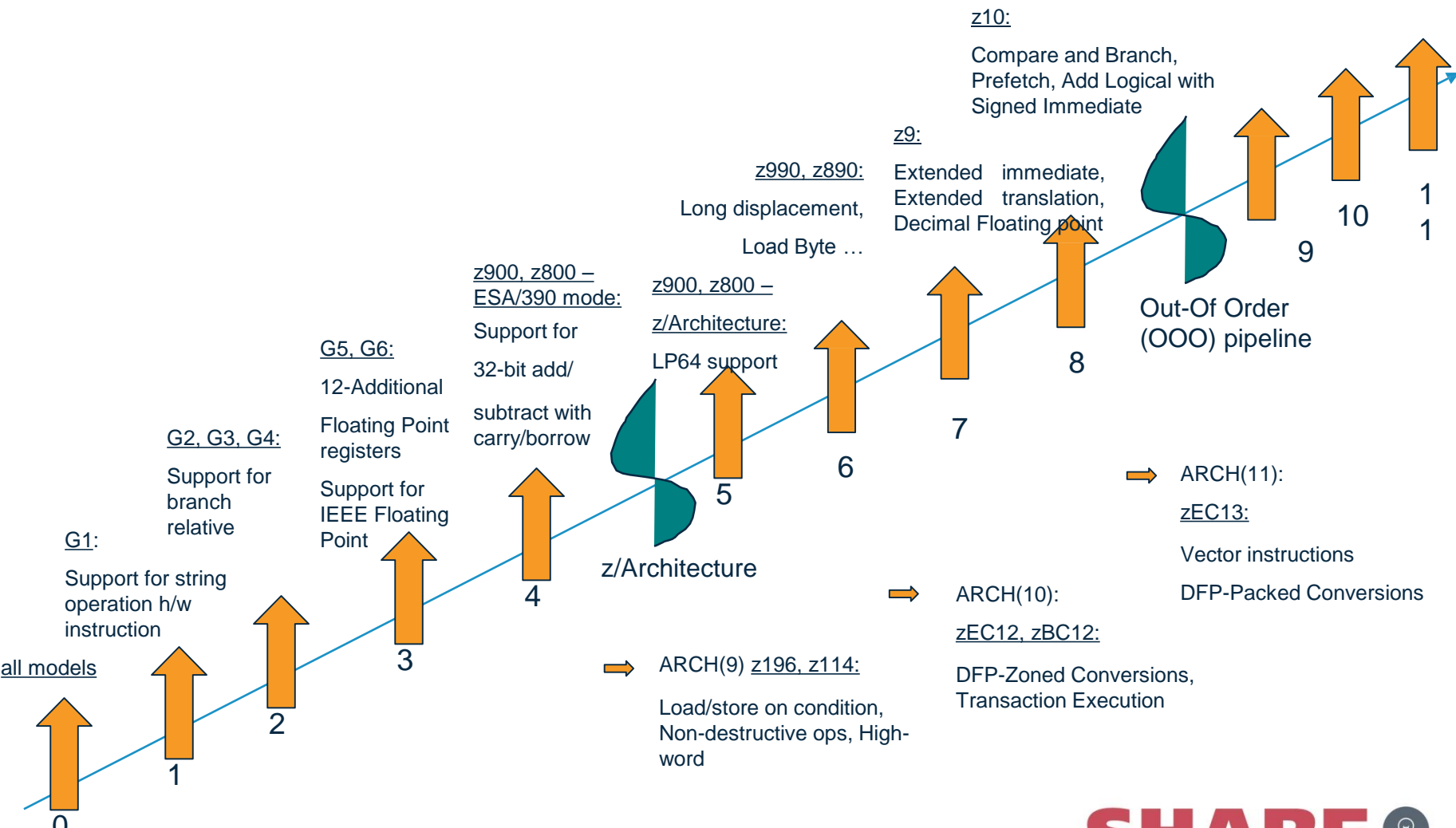- Make use of attributes and keywords

- OpenMP [C/C++]

Code for performance

- Keep it simple and concise
    - Good for both the programmer and the compiler to understand the code easily
- Don't ignore the compiler informational and warning messages, even if the program appears to work
- Attempts to be clever and produce "optimal" code might produce:
    - Code that is unreadable
    - Code that cannot be maintained
    - Code that performs worse than the straightforward solutions
    - Code that fails

ARCHitecture Option

- The ARCH option specifies the level of the hardware on which the generated code must run
  - C/C++ default is ARCH(8) for V2R2 and up
    - That's code customized for z10 EC and z10 BC
  - PL/I default is ARCH(7) for 4.5 and up
    - produces code that will run on z9 (or later) machines
  - LE 2.1 requires z9 (or later) machines

- **However:** you must set ARCH to the lowest level machine where your generated code will run
  - If you specify ARCH(n) and run the generated code on an ARCH(n-1) machine, you will most likely get an operation exception

ARCHitecture - Timeline

z10:

Compare and Branch, Prefetch, Add Logical with Signed Immediate

z9:

Extended immediate, Extended translation, Decimal Floating point

z990, z890:

Long displacement,

Load Byte …

Out-Of Order (OOO) pipeline

z900, z800 – ESA/390 mode:

Support for

32-bit add/

subtract with carry/borrow

z900, z800 – z/Architecture:

LP64 support

G5, G6:

12-Additional

Floating Point registers

Support for IEEE Floating Point

z/Architecture

ARCH(11):

zEC13:

Vector instructions

DFP-Packed Conversions

G2, G3, G4:

Support for branch relative

ARCH(10):

zEC12, zBC12:

DFP-Zoned Conversions, Transaction Execution

G1:

Support for string operation h/w instruction

ARCH(9) z196, z114:

Load/store on condition, Non-destructive ops, High-word

all models

0 1 2 3 4 5 6 7 8 9 10 11

SHARE in Orlando 2015

11

ARCH(9): Load/store on condition

consider this small program:

```
 2.0 |  test: proc returns( fixed bin(31) );
 3.0 |
 4.0 |    exec sql include sqlca;
 5.0 |
 6.0 |    dcl c fixed bin(31);
 7.0 |
 8.0 |    exec sql commit;
 9.0 |
10.0 |    if sqlcode = 0 then
11.0 |       c = 0;
12.0 |    else
13.0 |       c = -1;
14.0 |
15.0 |    return( c );
16.0 | end;
```

ARCH(9): Load/store on condition

- Under OPT(3) ARCH(8), the instructions after the call are:

```
0000CA  0DEF                        000008 |          BASR    r14,r15
0000CC  5800  D0F4                  000010 |          L       r0,<a1:d244:14>(,r13,244)
0000D0  A718  FFFF                  000010 |          LHI     r1,H'-1'
0000D4  EC06  0005  007E            000010 |          CIJNE   r0,H'0',@1L8
0000DA  4110  0000                  000010 |          LA      r1,0
0000DE                              000010 |  @1L8    DS      0H
0000DE  58E0  2000                  000015 |          L       r14,_addrReturns_Value(,r2,0)
0000E2  5010  E000                  000015 |          ST      r1,_shadow1(,r14,0)
```

#SHAREorg

SHARE is an independent volunteer-run information technology association
that provides education, professional networking and industry influence. 13

Copyright (c) 2015 by SHARE Inc. Except where otherwise noted, this work is licensed under
http://creativecommons.org/licenses/by-nc-sa/3.0/

ARCH(9): Load/store on condition

- under OPT(3) ARCH(9), the instructions after the call are:

```
0000CA  0DEF            000008 |         BASR    r14,r15
0000CC  A718  FFFF      000010 |         LHI     r1,H'-1'
0000D0  BF0F  D0F4      000010 |         ICM     r0,b'1111',<a1:d244:l4>(r13,244)
0000D4  58E0  2000      000015 |         L       r14,_addrReturns_Value(,r2,0)
0000D8  4100  0000      000010 |         LA      r0,0
0000DC  B9F2  8010      000010 |         LOCRE   r1,r0
0000E0  5010  E000      000015 |         ST      r1,_shadow1(,r14,0)
```

#SHAREorg

SHARE is an independent volunteer-run information technology association
that provides education, professional networking and industry influence. 14

Copyright (c) 2015 by SHARE Inc. Except where otherwise noted, this work is licensed under
http://creativecommons.org/licenses/by-nc-sa/3.0/

ARCH(9): Load/store on condition

- So, under ARCH(8), the code sequence was:
  - Load SQLCODE into r0
  - Load -1 into r1
  - Compare r0 (SQLCODE) with 0 and branch if NE to @1L8
  - Load 0 into r1
  - @1L8
  - Store r1 into the return value

- While under ARCH(9), the code sequence has no label and no branch:
  - Load -1 into r1
  - Load SQLCODE into r0 via ICM (so that CC is set)
  - Load 0 into r0
  - Load-on-condition r1 with r0 if the CC is zero (i.e. if SQLCODE = 0)
  - Store r1 into the return value

ARCH(10): DFP Zoned Conversion Facility

- This code converts a PICTURE array to FIXED BIN

```
pic2int: proc( ein, aus ) options(nodescriptor);

  dcl ein(0:100_000) pic'(9)9' connected;
  dcl aus(0:hbound(ein)) fixed bin(31) connected;
  dcl jx     fixed bin(31);

  do jx = lbound(ein) to hbound(ein); aus(jx) =
    ein(jx);
  end; end;
```

ARCH(10): DFP Zoned Conversion Facility

- Under ARCH(9), the heart of the loop consists of these 8 instructions

```
0058   F248   D098   1000        PACK      #pd580_1(5,r13,152),_shadow2(9,r1,0)

005E   C020   0000   0021        LARL      r2,F'33'

0064   D204   D0A0   D098        MVC       #pd581_1(5,r13,160),#pd580_1(r13,152)

006A   4110   1009               LA        r1,#AMNESIA(,r1,9)

006E   D100   D0A4   200C        MVN       #pd581_1(1,r13,164),+CONSTANT_AREA(r2,12)

0074   F874   D0A8   D0A0        ZAP       #pd582_1(8,r13,168),#pd581_1(5,r13,160)

007A   4F20   D0A8               CVB       r2,#pd582_1(,r13,168)

007E   502E   F000               ST        r2,_shadow1(r14,r15,0)
```

ARCH(10): DFP Zoned Conversion Facility

- While under ARCH(10), it consists of 9 instructions and uses DFP in several of them – but since only the ST and the new CDZT refer to storage, the loop runs more than 66% faster

```
0060    EB2F   0003 00DF          SLLK    r2,r15,3

0066    B9FA   202F               ALRK    r2,r15,r2

006A    A7FA   0001               AHI     r15,H'1'

006E    B9FA   2023               ALRK    r2,r3,r2

0072    ED08   2000 00AA          CDZT    f0,#AddressShadow(9,r2,0),b'0000'

0078    B914   0000               LGFR    r0,r0

007C    B3F6   0000               IEDTR   f0,f0,r0

0080    B941   9020               CEDTR   r2,b'1001',f0
```

ARCH(11): Vector Instruction Facility

- With ARCH(11), the vector instruction facility is used to inline it as

```
E700   E000 0006          VL       v0,+CONSTANT_AREA(,r14,0)
E740   E010 0006          VL       v4,+CONSTANT_AREA(,r14,16)

                    @1L2   DS       0H

A74E   0010              CHI      r4,H'16'

4150   0010              LA       r5,16

B9F2   4054             LOCRL    r5,r4

B9FA   F0E2             ALRK     r14,r2,r15

E725   E000 0037         VLL      v2,r5,_shadow1(r14,0)

E722   0180 408A         VSTRC    v2,v2,v0,v4,b'0001',b'1000'

E7E2   0001 2021         VLGV     r14,v2,1,2

EC5E   000D 2076         CRJH     r5,r14,@1L3
```

ARCH(11): Vector Instruction Facility

```
foo (double * restrict p,
     double * restrict q,
     double * restrict r,
     int n) {
  for (int i=0; i<n; i++) {
    p[i]=p[i]+q[i]*r[i];
}
return 0;
}

xlC -c report.C -qlist=./ -qHOT -qARCH=11 -qVECTOR -qphaseid  -qfloat=ieee
```

Complete your session evaluations online at www.SHARE.org/Orlando-Eval

19

20

ARCH(11): Vector Instruction Facility

```
 000160                           000005 |         @1L41    DS        0H
                                  000006 |       *         p[i]=p[i]+q[i]*r[i];
 000160   E70E   3000   0006      000006 |                 VL        v0,@V.(r{double})0(r14,r3,0)
 000166   E72E   2000   0006      000006 |                 VL        v2,@V.(q{double})1(r14,r2,0)
15650ZOS V2.2 z/OS XL C++                                       report.C: foo(_bfp
doub...)   08/10/15 10:40:42                4

 OFFSET OBJECT CODE            LINE#  FILE#     P S E U D O   A S S E M B L Y   L I S T I N G

 00016C   E74E   1000   0006      000006 |                 VL        v4,@V.(p{double})2(r14,r1,0)
 000172   E702   0000   30E7      000006 |                 VFM       v0,v2,v0,b'0011',b'0000'
 000178   E704   0000   30E3      000006 |                 VFA       v0,v4,v0,b'0011',b'0000'
 00017E   E70E   1000   000E      000006 |                 VST       v0,@V.(p{double})2(r14,r1,0)
 000184   41E0   E010             000000 |                 LA        r14,#AMNESIA(,r14,16)
 000188   A746   FFEC             000005 |                 BRCT      r4,@1L41
```

ARCHitecture Option

- The wonderful feature of the ARCH option is that no code changes are required by you

- In all of the above examples, the compiler
  - figured out where it could exploit the option
  - and then did all the work

HGPR Option

- Stands for High half of 64-bit General Purpose Register
- Permitted to exploit 64-bit GPRs in 32-bit programs
  - Compiler can now make use of
    - The 64-bit version of the z/Architecture instructions
      - Improves run-time performance for source programs that utilize 64-bit types in 32-bit amode, e.g. long long
- PRESERVE sub-option
  - Save/re-store in prolog/epilog the high halves of used GPRs
  - Only necessary if the caller is not known to be compiler-generated code
- Default is NOHGPR(NOPRESERVE)
- Metal C defaults to HGPR(PRESERVE)

FLOAT(AFP) Option

- Additional Floating-Point (AFP) registers were added to ESA/390 models

- AFP sub-option enable use of the full set (16) of FPRs

- VOLATILE sub-option

  - FPR8 – FPR15 is considered volatile

    - i.e. compiler will not expect they're preserved by any called program

  - No longer required for CICS TS V4.1 or newer

- Default is AFP(NOVOLATILE)

OPTIMIZE Option

- The OPT option controls how much, or even if at all, the compiler tries to optimize your code
    - A trade-off between compile-time vs. execution-time

- NOOPT/OPT(0):
    - The compiler simply translates your code into machine code
    - Generated code could be large and slow
    - Good choice for:
        - Matching code generated with written source code
            - for the purpose of debugging a problem
        - Reducing compile time
    - Terrible choice if you care about run-time performance

OPTIMIZE Option (cont'd)

- When optimizing, the compiler will improve, often vastly, the code it generates by, for example
    - Keeping intermediate values in registers
    - Moving code out of loops
    - Merging statements
    - Reordering instructions to improve the instruction pipeline
    - Inlining functions

- Require more CPU and REGION during compilation

OPTIMIZE Option (cont'd)

- OPT(2):
  - Start enabling the optimizer
  - A balance between compile speed and code quality
- OPT(3):
  - Optimizer much more aggressive
  - Tips balance towards code quality over compile speed
  - C/C++ compiler will alter other options defaults:
    - ANSIALIAS, IGNERRNO, STRICT, etc
- The C/C++ and PL/I compilers use the same optimizing backend
  - But there are differences in what the OPT sub-options does

OPTIMIZE Option (cont'd)

Base optimization level - xlc utility option	Base optimization level - z/OS batch option	Additional options implied by optimization level	Additional recommended options
-O0	OPT(0) or NOOPT	None	ARCH(n)
-O2	OPT(2)	MAXMEM(*)	ARCH (n) INLINE (to tune inlining) TUNE(n)
-O3	OPT(3)	NOSTRICT MAXMEM(*)	ARCH(n) TUNE(n)
-O4	OPT(3) HOT IPA(LEVEL(1))	All of OPT(3) plus: HOT IPA	ARCH(n) TUNE(n) PDF
-O5	OPT(3) HOT IPA(LEVEL(2))	All of -O4 plus: IPA(LEVEL(2))	ARCH(n) TUNE(n) PDF

Other C/C++ Options Related to OPT

- HOT option

  - High-Order loop analysis and Transformations

    - More aggressive optimization on the loops

    - Requires OPT(2) or higher

- IPA option

  - Inter-Procedural Analysis

    - Optimization decisions made based on the entire program

    - 3 sub-levels to control aggressiveness

    - Requires OPT(2) or higher

  - PDF sub-option

    - Profile Directed Feedback

      - Sample program execution to help direct optimization

      - Requires a training run with representative data

IPA Option [C/C++] (cont'd)

IPA compile

file1.c → xlc → file1.o

file2.c → xlc → file2.o

file3.c → xlc → file3.o

libraries

xlc → IPA(LINK)

binder

executable
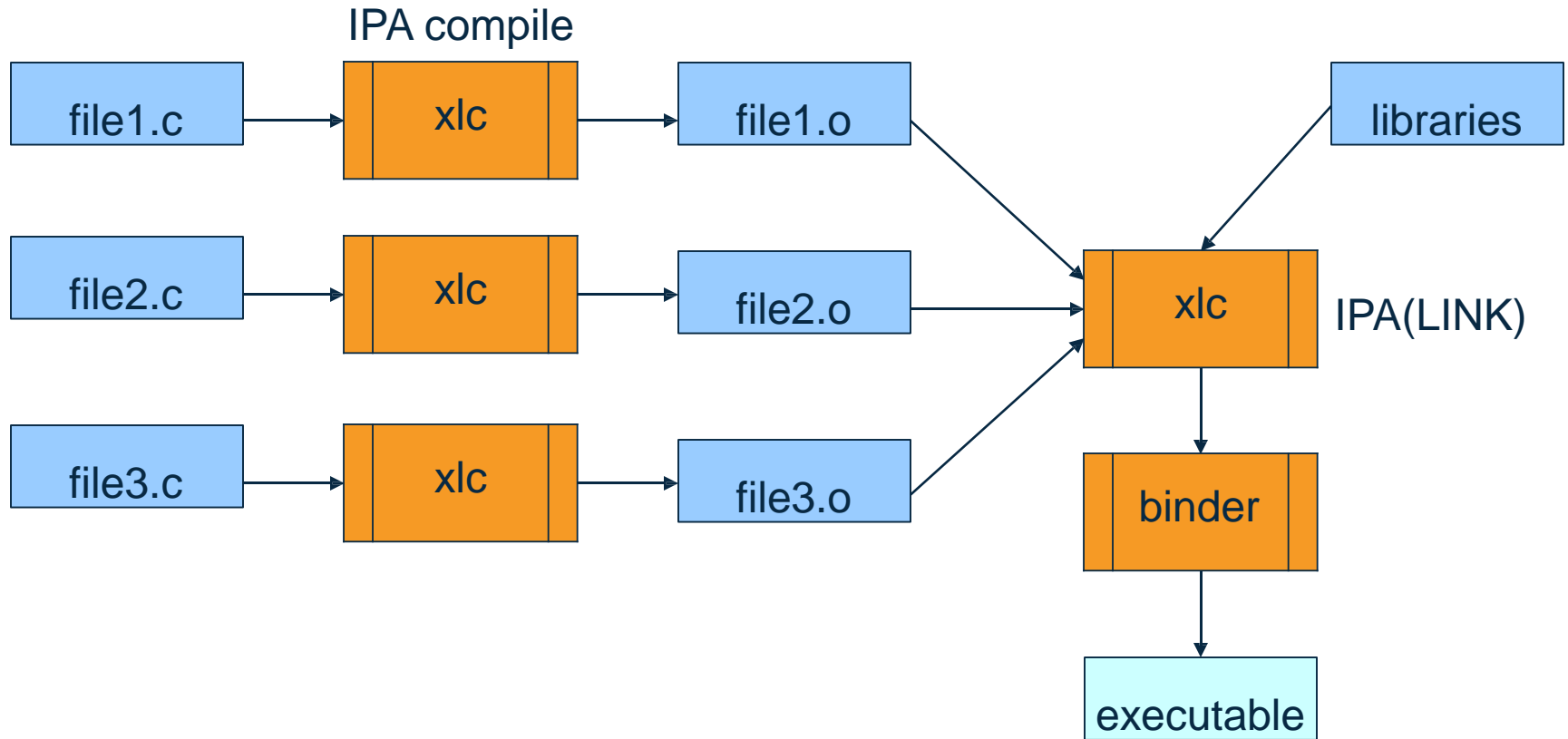
IPA PDF Sub-Option [C/C++]

PDF1:
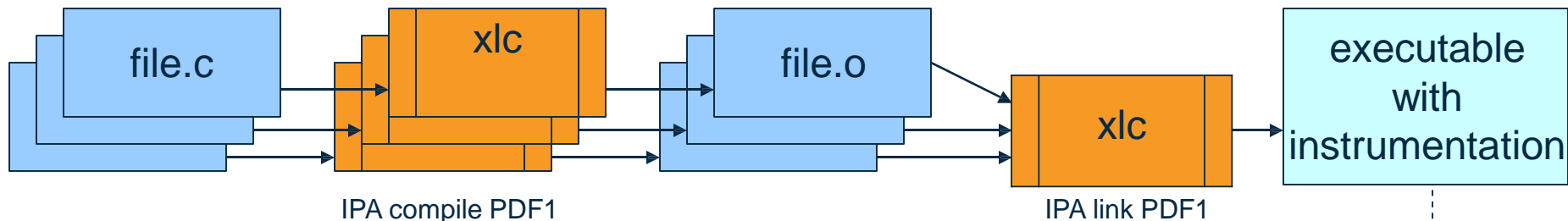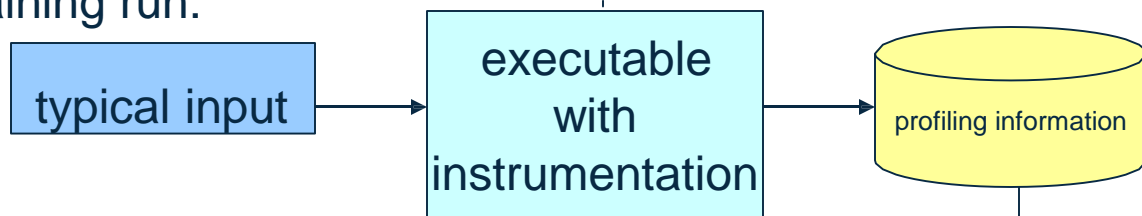
file.c → **xlc** → file.o → **xlc** → executable with instrumentation

IPA compile PDF1   IPA link PDF1

Training run:

typical input → executable with instrumentation → profiling information

PDF2:

file.o → **xlc** → PDF optimized executable (w/o instrumentation)

IPA link PDF2
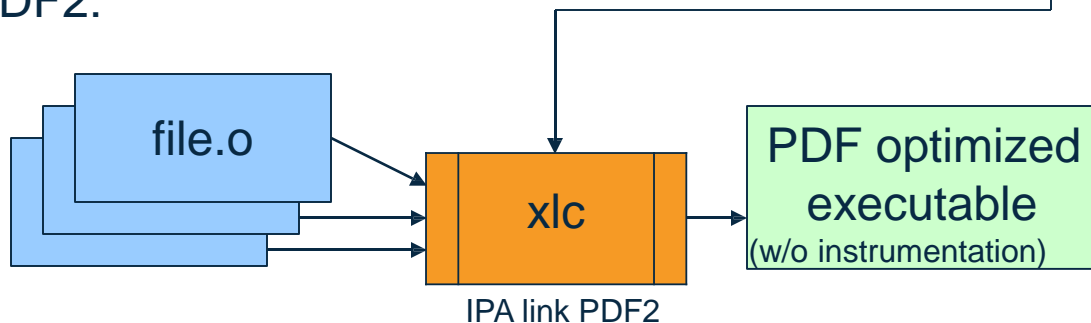
ANSIALIAS Option [C/C++]

- Optimizer presumes pointers can point only to objects of the same type

  - The simplified rule is that you cannot safely dereference a pointer that has been cast to a type that is not closely related to the type of what it points at

    - The ISO C and C++ standards define the closely related types

- If this assumption is false, wrong code could be generated

  - The INFO(ALS) option might able to help you find potential violation of the ANSI type-based aliasing rule

- OPT(3) defaults to ANSIALIAS

- OPT(2) defaults is NOANSIALIAS

IGNERRNO Option [C/C++]

- Informs the compiler that the program is not using errno
- Allows the compiler more freedom to explore optimization opportunities for certain library functions
  - For example: sqrt
- Need to include the system header files to get the full benefit

- OPT(3) defaults to IGNERRNO
- NOOPT and OPT(2) defaults are NOIGNERRNO

LIBANSI Options [C/C++]

- Indicates the name of an ANSI C library function are in fact ANSI C library functions and behave as described in the ANSI standard

- The optimizer can generate better code based on existing behavior of a given function
    - E.g. whether or not a particular library function has any side effects

- Provides additional benefits when used in conjunction with IGNERRNO

- Defaults is NOLIBANSI

NOTHREADED Option [C/C++]

- For user to assert their application is single-threaded
- Allows for non-thread-safe transformations be performed

- Defaults is THREADED

NOSTRICT Option [C/C++]

- Allows the optimizer to alter the semantics of a program
  - Performing code motion and scheduling on computations such as loads and floating-point computations that may trigger an exception
  - Relax conformance to IEEE rules
  - Reassociating floating-point expressions

- OPT(3) defaults is NOSTRICT
- NOOPT and OPT(2) defaults are STRICT

NOSTRICT_INDUCTION Option [C/C++]

- Asserts to the compiler the induction (loop counter) variables do not overflow or wrap-around
    - Use STRICT_INDUCTION only if your program logic has such intent
- Only affects loops which have an induction variable declared with a different size than a register

- Default is NOSTRICT_INDUCTION
    - Except with the c99 invocation command on USS

XPLINK Option [C/C++]

- XPLINK stands for eXtra Performance LINKage
  - A modern linkage convention that is 2.5 times more efficient than the conventional linkage conventions
  - We have seen some programs improved by 30%
  - XPLINK and non-XPLINK parts can work across DLL and fectch() boundaries
    - Must tell compiler about this, so the (expensive) switching code get executed
    - If your application contains few switches, then mixing will still be beneficial
- Defaults:
  - ILP32: NOXPLINK
  - LP64: XPLINK

REDUCE and RESEXP Options [PL/I]

- ## REDUCE option

    - Specifies that the compiler is permitted to reduce an assignment of a null string to a structure into a simpler operation

        - Even if that means padding bytes might be overwritten or zerored out

- ## RESEXP option

    - Specifies that the compiler is permitted to evaluate all restricted expressions at compile time even if this would cause a condition to be raised and the compilation to end with S-level messages

RULES(NOLAXCTL) Option [PL/I]

- Specifies that the compiler disallows a CONTROLLED variable to be declared with a constant extent and yet to be allocated with a differing extent

- To allocate a CONTROLLED variable with a variable extent, that extents must be declared either with an asterisk or with a non-constant expression.

- When the compiler sees a reference to a structure, or to any member of that structure, it knows the lengths, dimensions or offsets of the fields in it

DEFAULT Sub-Option
CONNECTED REORDER NOOVERLAP

- CONNECTED sub-option
  - Compiler presumes application never passes nonconnected parameters

- REORDER sub-option
  - Indicates that the ORDER option is not applied to every block, meaning the compiler doesn't have to insure that variables referenced in ON-units (or blocks dynamically descendant from ON-units) have their latest values

- NOOVERLAP sub-option
  - Compiler presumes the source and target in an assignment do not overlap

COMPACT Option [C/C++]

- Compiler favors optimizations that tend to limit the growth of the code

- Depending on your specific program, the object size may increase or decrease and the execution time may increase or decrease

- Default is NOCOMPACT

- PL/I effectively always has NOCOMPACT on

INLINE Option [C/C++]
DEFAULT(INLINE) Option [PL/I]

- Inlining eliminates the overhead of the function call and linkage, and also exposes the function's code to the optimizer

- Too much inlining can increase the size of the program

- AUTO sub-option [C/C++]
  - Inliner runs in automatic mode
  - Threshold sub-option
    - Maximum relative size of a subprogram to inline
  - LIMIT sub-option
    - Maximum relative size a subprogram can grow before auto-inlining stops

UNROLL Option

- Instructs the compiler to perform loop unrolling
- It replicates a loop body multiple times, and adjusts the loop control code accordingly
- It increases code size in the new loop body

- Auto sub-option
    - Compiler decides via heuristics the appropriate candidate and amount of unrolling

Make Use Of Built-in Functions

- Library function example:
  - Less efficient comparison on a loop
    ```
    int i, a[1000], b[1000];

    …

    for (i = 0; i < 1000; ++i) if (a[i] != b[i])
                             break;
    if (i == 1000)
      /* arrays are equal */
    ```
  - More efficient comparison with a memcmp() library function
    ```
    int a[1000], b[1000];

    …

    if (!memcmp (a, b, sizeof(a)))
       /* arrays are equal */
    ```

Make Use Of Built-in Functions (cont'd)

- Hardware built-in function example
    - A naive implementation of population count

```
unsigned long popcount(unsigned long op) {
   unsigned long count = 0;
   unsigned long bit = 1;
   for (int i = 0; i < 64; i++) {
     if (op & bit)
                  count++;
     bit = bit << 1;
   }
   return count;
}
```

    - with   popcnt() hardware built-in function

```
unsigned long    popcnt(unsigned long op)
```

        - Available from ARCH(9)
            - A single POPCNT instruction
            - Or as POPCNT built-in function in PL/I

Make Use Of #pragmas [C/C++]

- Provides more details about your code to help the optimizer
  - #pragma execution_frequency (C++only)
    - Marks program source code that you expect will be either very frequently or very infrequently executed
  - #pragma isolated_call
    - Lists functions that have no side effects (that do not modify global storage)
- For fine-grained control
  - #pragma inline
    - Hint to the compiler to inline this frequently used function
  - #pragma noinline
    - Prevents a function from being inlined
  - #pragma unroll
    - Informs the compiler how to perform loop unrolling on the loop body that immediately follows it

Make Use of Attributes & Keywords [C/C++]

- Provides more details about your code to help the optimizer
  - restrict keyword
    - Use with ASSERT(RESTRICT) to indicate disjointed pointers
      - Defaults is ASSERT(RESTRICT)
    - Two restrict qualified pointers, declared in the same scope, designate distinct objects and thus shouldn't alias each other
    - RESTRICT option (C only) can also be used to indicates to the compiler that pointer parameters in all functions or in specified functions are disjoint
      - Defaults is NORESTRICT

- For fine-grained control
  - inline keyword
    - Hint to the compiler to inline this frequently used function
  - always_inline function attribute
    - Instructs the compiler to inline a function

Make Use of Attributes & Keywords [PL/I]

- Use RETURNS(BYVALUE) for items that can be returned in registers (such as FIXED BIN and FLOAT)

- Use the BYVALUE attribute on parameters that are input-only and which can be passed in registers

- Use the INONLY, OUTONLY, and NONASSIGNABLE attributes on parameters and in ENTRY declares

- Routines with OPTIONS(LINKAGE(OPTLINK)) will outperform those with OPTIONS(LINKAGE(SYSTEM))

Make Use of Attributes & Keywords [PL/I]

- You should always fully prototype all ENTRY declarations

- Specify BYADDR/BYVALUE and (NON)ASGN for each parameter

- And specify (NON)CONNECTED for each array parameter

- Also specify BYADDR/BYVALUE for the RETURNS

- Also include an OPTIONS attribute and specify therein the LINKAGE as well as NODESCRIPTOR options (as appropriate)

OpenMP API 3.1 [C/C++]

- Industry-standard API designed to create portable C/C++ applications to exploit shared-memory parallelism
- Users can create or migrate parallel applications to take advantage of the multi-core design of modern processors
- Consists of a collection of compiler directives and library routines
- New SMP option to allow OpenMP parallelization directives to be recognized
  - Only supported in 64-bit
  - Executable must be run under USS
  - Thread-safe version of standard library must be used inside the parallel regions
  - Not supported with Metal C

Declare your variables

- A common sign in Texas:

    – **Trespassers will be prosecuted or shot**

- Those who don't declare their variables deserve the same fate

- Use the RULES(NOLAXDCL) compiler option to enforce this in PL/I

Declare your variables with good names

- Generally, you should not name a variable after its type,

- i.e. do not code the following

   DCL BASED_FB15   FIXED BIN(15) BASED; DCL
      1 ELEMENT_REC BASED,

         2 NEXT_PTR   PTR,
         2 PREV_PTR   PTR,
         2 DATA, ….

- Because this name becomes meaningless if PTR becomes OFFSET

Declare your variables with attributes

- Simply declaring the name is not good

- i.e. don't code: DCL RC;

- Because then RC is FLOAT DEC(6) when FIXED BIN(31) was probably what was wanted.

- The compiler will issue warning message IBM1215 for such declares – or message IBM1216 if part of a structure

Declare your variables with attributes

- A common way this error occurs is in code such as

    – **DCL RC1, RC2 FIXED BIN(31) INIT(0);**

- Enterprise PL/L issues message IBM1215 saying that RC1 is declared without any attributes

- And like the old compiler, Enterprise PL/I will give RC1 the attributes FLOAT DEC(6) – not FIXED BIN

- The declare above is not the same as

    – **DCL (RC1, RC2) FIXED BIN(31) INIT(0);**

Declare your variables with attributes

- Some customer code contained this code

```
DCL
  PARDIASE CHAR(20),
  1 INDIASE1 BASED (PTPDIASE),
    2 C1CODIA   CHAR(1),
    2 C1FECDI   DEC FIXED(9),
    2 C1DIADI   CHAR(9),
    2 C1ABRDI   CHAR(3),
    2 C1RESDI;
```

- Here the compiler issues the message IBM1216 saying that C1RESDI is declared without any attributes

- Again, C1RESDI will get the attributes FLOAT DEC(6)

Declare your variables with attributes

- However, this means the structure needs 22 bytes

```
DCL
  PARDIASE CHAR(20),
  1 INDIASE1 BASED (PTPDIASE),
    2 C1CODIA   CHAR(1),
    2 C1FECDI   DEC FIXED(9),
    2 C1DIADI   CHAR(9),
    2 C1ABRDI   CHAR(3),
    2 C1RESDI;
```

- And then this later bit of code overwrites 2 bytes of storage

```
PTPDIASE = ADDR(PARDIASE); INDIASE1 = '';
```

- This leads to a protection exception in some circumstances, and remember, this is a user error, not a compiler error

Declare your variables with sensible attributes

- You will get warning message IBM1091 with text

    – **FIXED BIN precision less than storage allows**

- If you declare (or use in a built-in)

    – **SIGNED FIXED BIN with precision other than 7, 15, 31 or 63**

    – **UNSIGNED with precision other than 8, 16, 32 or 64**

- Most users would think this couldn't possibly be an issue for them

Declare your variables with sensible attributes

- But this banking code copies an array to a new array twice as large

```
40.1    UBSEMB:PROC(ACCOUNT_TABLE) REORDER;

42.1      DCL 1 ACCOUNT_TABLE(*)   CONTROLLED,
43.1            2 CUSTOMER_NAME        CHAR(120),
44.1            2 ACCT_INSTR_NUMBER  CHAR(17),
45.1            2 ACCT_INSTR_CODE   CHAR(8),
46.1            2 ORIGINAL_BLNCE_AMT CHAR(9),
47.1            2 DATE_OF_LAST_TXN,
48.1             3 YEAR               CHAR(4),
49.1             3 MONTH              CHAR(2),
50.1             3 DAY                CHAR(2);


55.1      DCL   NEW_SIZE           FIXED BIN(5) INIT(0);
56.1      DCL   OLD_SIZE           FIXED BIN(5) INIT(0);


57.1      DCL   RECORD_NO          FIXED BIN(5) INIT(1);


58.1      DCL 1 TEMP_TABLE(*)      CONTROLLED,


59.1            2 CUSTOMER_NAME       CHAR(120),


60.1            2 ACCT_INSTR_NUMBER   CHAR(17),

61.1            2 ACCT_INSTR_CODE     CHAR(8),

62.1            2 ORIGINAL_BLNCE_AMT CHAR(9),
```

Declare your variables with sensible attributes

- ## Via this small bit of code

```
68.1      NEW_SIZE = HBOUND(ACCOUNT_TABLE.CUSTOMER_NAME,1) * 2;
69.1      ALLOCATE TEMP_TABLE(NEW_SIZE);
70.1      TEMP_TABLE(*) = '';
71.1      OLD_SIZE = HBOUND(ACCOUNT_TABLE.CUSTOMER_NAME,1); DO
72.1      RECORD_NO = 1 TO OLD_SIZE;
73.1       TEMP_TABLE(RECORD_NO) = ACCOUNT_TABLE(RECORD_NO);
74.1      END;
75.1      FREE ACCOUNT_TABLE;
76.1      ALLOCATE ACCOUNT_TABLE(NEW_SIZE); ACCOUNT_TABLE =
77.1      TEMP_TABLE;
78.1      FREE TEMP_TABLE;
79.1      FREE TEMP_TABLE;
        END; /*UBSEMB*/
```

- ## And it abends

- ## Only because the customer ignored message IBM1091 flagging that a variable was declared as FIXED BIN(5) (when 15 was almost certainly intended)

Declare your variables with sensible attributes

```
40.1    UBSEMB:PROC(ACCOUNT_TABLE) REORDER;

42.1     DCL 1 ACCOUNT_TABLE(*)   CONTROLLED,
43.1          2 CUSTOMER_NAME        CHAR(120),
44.1          2 ACCT_INSTR_NUMBER    CHAR(17),
45.1          2 ACCT_INSTR_CODE    CHAR(8),
46.1          2 ORIGINAL_BLNCE_AMT CHAR(9),
47.1          2 DATE_OF_LAST_TXN,
48.1           3 YEAR               CHAR(4),
49.1           3 MONTH              CHAR(2),
50.1           3 DAY                CHAR(2);


55.1     DCL   NEW_SIZE            FIXED BIN(5) INIT(0),
56.1     DCL   OLD_SIZE            FIXED BIN(5) INIT(0);

57.1     DCL   RECORD_NO           FIXED BIN(5) INIT(1);

58.1     DCL 1 TEMP_TABLE(*)       CONTROLLED,

59.1          2 CUSTOMER_NAME      CHAR(120),

60.1          2 ACCT_INSTR_NUMBER  CHAR(17),

61.1          2 ACCT_INSTR_CODE    CHAR(8),

62.1          2 ORIGINAL_BLNCE_AMT CHAR(9),

63.1          2 DATE_OF_LAST_TXN
```

Describe your interfaces

- This starts with how you declare external routines

- Do not declare them without a parameter list as in

    - **DCL A EXT ENTRY;**

- This lets you pass any number of arguments of any type to this routine without the compiler being able to check your code

- The compiler would quietly accept all of these

    - **CALL A;**
    - **CALL A(TIMESTAMP);**
    - **CALL A(2, JJJJ);**

Describe your interfaces

- Be accurate – if the routine has no parameters, say so

  - **DCL A EXT ENTRY();**

- Or if the routine should receive one string, declare it as

  - **DCL A EXT ENTRY(CHAR(*));**

- Now the compiler can flag bad calls of this routine

- And if a string parameter must have a certain length, say that:

  - **DCL A EXT ENTRY(CHAR(17));**

- But then you need to be especially on watch for messages about "dummy" arguments

Recap

- Let the compiler work for you by telling it
  - The hardware to exploit
  - The importance of compile-time vs. execution performance
  - More precise details about the source code
  - Sensitiveness of module size

- Work together with the compiler
  - Writing good code
  - Make use of BIFs and #pragmas
  - Exploit the language features
  - Tell the compiler what you know

Additional Reading Materials

- z/OS C/C++ Programming Guide
  - Part 5. Performance optimization
  - http://pic.dhe.ibm.com/infocenter/zos/v2r1/topic/com.ibm.zos.v2r1.cbcpx01/cbc1p2399.htm

- Enterprise PL/I for z/OS Programming Guide
  - Chapter 13. Improving performance
  - http://publibfp.boulder.ibm.com/epubs/pdf/ibm4pg03.pdf

Quick Survey

- Users of:

  - PL/I

  - C/C++

  - NOOPTIMIZE/OPTIMIZE(0), OPTIMIZE(2), OPTIMIZE(3)

  - ARCH(7), ARCH(8), ARCH(9), ARCH(10)

  - C/C++ only:

    - TUNE

    - LP64

    - PDF

    - HOT

    - IPA

Questions?

- Connect with us
  - Email me at visdav @ca.ibm.com
  - Rational Café - the compilers user community & forum
    - C/C++: http://ibm.com/rational/community/cpp
    - PL/I: http://ibm.com/rational/community/pli
  - RFE community – for feature requests
    - C/C++: http://www.ibm.com/developerworks/rfe/?PROD_ID=700
    - PL/I: http://www.ibm.com/developerworks/rfe/?PROD_ID=699
  - Product Information
    - C/C++: http://www-03.ibm.com/software/products/us/en/czos
    - PL/I: http://www-03.ibm.com/software/products/en/plicompfami

Thank You!