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

Dickson Chau
IBM

August 7, 2014
Session 16091
WHAT ...

- does good application performance mean to you?
  - Fast Execution Time
  - Short Compile Time
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
System z Models

- **z900** – Full 64-bit z/Architecture
- **z990** – Superscalar CISC pipeline
- **z9** – System level scaling
- **z10** – Deep Pipeline, Arch. extensions
- **z196** – Out-Of-Order (OOO), Additional Architectural Extensions
- **zEC12** – OOO+, Architectural Extensions, Enablement for new Software Paradigms

**Technology + Design Optimizations**
- Provides higher performance & capacity
- Maintains unmatched reliability
- Supports Peak workload 24x7
- At similar power constraints
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++:
    • ANSI_ALIAS
    • IGNERRNO
    • LIBANSI
    • NOTHREADED
    • NOSTRICT
    • STRICT_INDUCTION
    • XPLINK
  • PL/I:
    • REDUCT
    • RESEXPT
    • RULES(NOLAXCTL)
    • DEFAULT(CONNECTED REORDER NOOVERLAP)
Utilize Compiler Options (cont’d)

- Controls load module size:
  - COMPACT [C/C++]
  - INLINE [C/C++]
  - DEFAULT(INLINE) [PL/I]
  - UNROLL
ARCHitecture Option

• The ARCH option specifies the level of the hardware on which the generated code must run
  • C/C++ default – is ARCH(7)*
    • produces code that will run on z9 machines
  • PL/I default – is ARCH(6)
    • produces code that will run on z990/z980 machines
• 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

* new default in z/OS XL C/C++ V2R1. Default architecture is ARCH(5) for all versions before this. ARCH(5) produces code that runs on z900 and newer
ARCHitecture - Timeline

- **ARCH(10):**
  - zEC12, zBC12:
    - DFP-Zoned Conversions, Transaction Execution

- **z/Architecture**
  - Out-Of Order (OOO) pipeline

- **z10:**
  - Compare and Branch, Prefetch, Add Logical with Signed Immediate

- **z9:**
  - Extended immediate, Extended translation, Decimal Floating Point

- **z900, z890:**
  - Long displacement, Load Byte …

- **z900, z800 – ESA/390 mode:**
  - Support for 32-bit add/subtract with carry/borrow

- **z900, z800 – z/Architecture:**
  - LP64 support

- **z196, z114:**
  - Load/store on condition, Non-destructive ops, High-word

- **ARCH(9):**
  - DFP-Zoned Conversions, Transaction Execution

- **G5, G6:**
  - 12-Additional Floating Point registers
  - Support for IEEE Floating Point

- **G2, G3, G4:**
  - Support for branch relative
  - Support for string operation h/w instruction

- **G1:**
  - Support for string operation h/w instruction
ARCH(9): Load-on-condition Example

```c
int bar(void);
int foo(void) {
    return ( (bar()==2) ? 1 : -1);
}

> xlc -c -O2 -qarch=8 -qlist loc.c
...

> xlc -c -O2 -qarch=9 -qlist loc.c
...```

```
ARCH(9): Load-on-condition Example
```
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
    - The High-Word Facility [with ARCH(7) or above]
      - Can be viewed as having an additional 16 GPRs
- 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(NOVOLTILE)
  - [C/C++] for ARCH(3) or higher
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
Other C/C++ Options Related to OPT

• HOT option
  • High-Order loop analysis and Transformations
    • More aggressive optimization on the loops
    • Requires to use with 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 \(\rightarrow\) xlc \(\rightarrow\) file1.o

file2.c \(\rightarrow\) xlc \(\rightarrow\) file2.o

file3.c \(\rightarrow\) xlc \(\rightarrow\) file3.o

IPA(LINK)

xlc \(\rightarrow\) binder

libraries

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
- Has no effect to NOOPT/OPT(0)
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
  • Like, 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 fetch() 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
      • *Padding bytes may be 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 NOOVERRIDE

• 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 maintain variables in that block referenced in ON-units (or blocks dynamically descendant from ON-units) have their latest values

• NOOVERRIDE sub-option
  • Compiler presumes the source and target in an assignment does 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
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
Code for Performance

- Writing good code
- Make use of built-in functions
- Make use of #pragmas [C/C++]
- Make use of attribute and keyword [C/C++]
- OpenMP [C/C++]
Writing Good Code

- 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
Make Use Of Built-in Functions

• Library function example:
  • Less efficient comparison on a loop
    
    ```c
    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
    
    ```c
    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
    ```
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 (C only)
    - 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 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(RESTRIC) to indicate disjointed pointers
      - *Defaults is ASSERT(RESTRIC)*
    - 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 NORESTRIC*
  - 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
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
OpenMP API 3.1 Example [C/C++]

```c
int bar(void) {
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        // executed in parallel by a # of threads
        ...
    }
}
```
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
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
References

• Visda Vokhshoori, Make Your C/C++ and PL/I Code FLY With the Right Compiler Options, SHARE Boston, Aug. 2013

• Peter Elderon, Make Your C/C++ and PL/I Code FLY With the Right Compiler Options, SHARE San Francisco, Feb. 2013
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
  • 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
• Product Information
  • C/C++: http://www-03.ibm.com/software/products/us/en/czos

Thank You!