#SHAREorg

SHARE
Technology · Connections · Results

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

Visda Vokhshoori/Peter Elderon
IBM Corporation

Session 13790

Insert Custom Session QR if Desired.

SHARE in Boston

WHAT does good application performance mean to you

- Fast execution time
- Short compile time
- Small load module size

HOW do you achieve good application performance

- Use best compiler option
- Write good code
- Install newer hardware

Options to consider for optimization

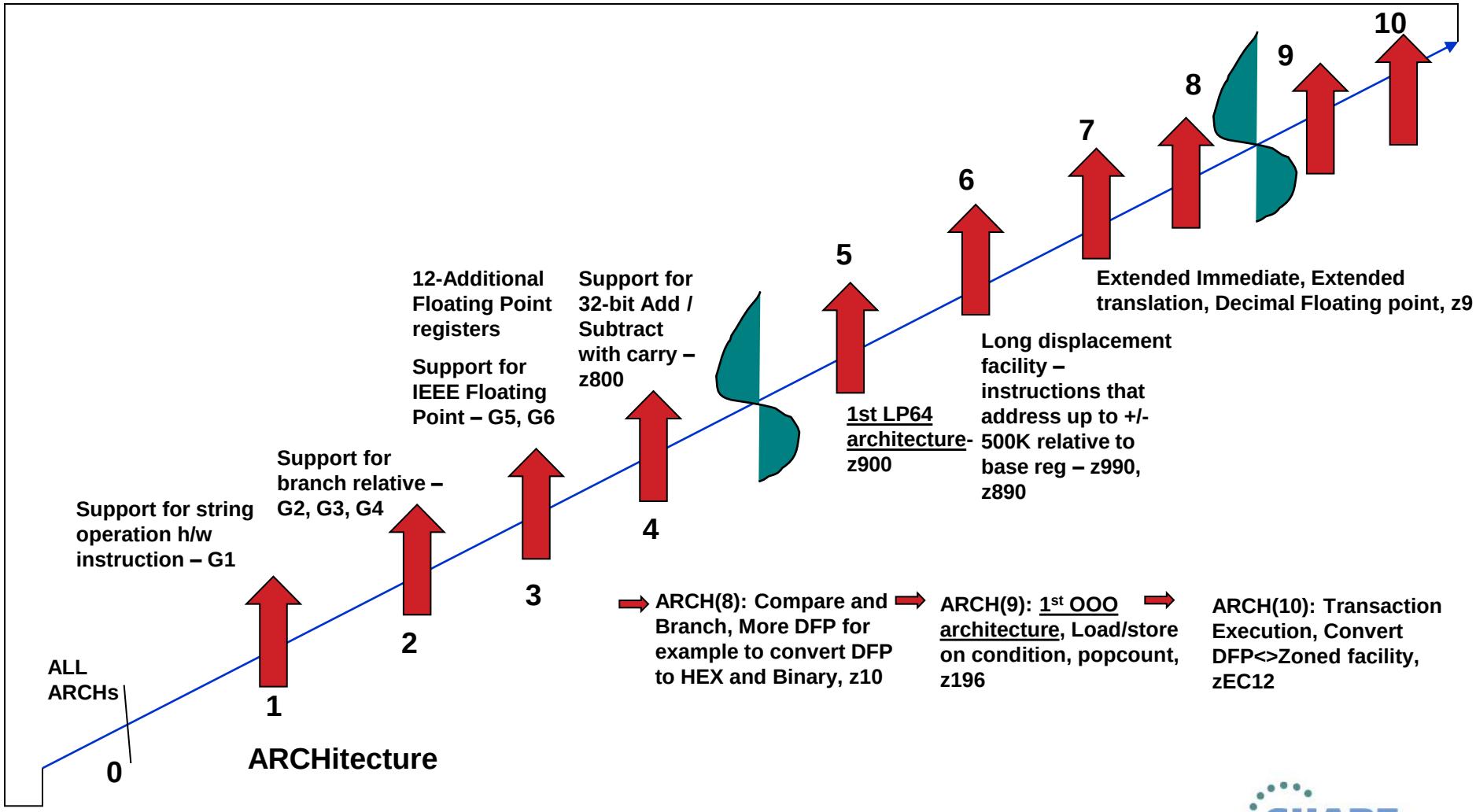
- ARCHitechture
- OPTIMIZE
- INLINE
- UNROLL
- FLOAT(AFP)
- HGPR
- [PL/I] REDUCE
- [PL/I] RESEXP
- [PL/I] RULES(NOLAXCTL)
- [PL/I] DEFAULT(REORDER NOOVERLAP CONNECTED)
- [C/C++] XPLINK
- [C/C++] COMPACT
- [C/C++] STRICT_INDUCTION
- [C/C++] Profile Directed Feedback
- [C/C++] Inter-Procedural Analysis

ARCHitecture

- The ARCH option specifies the level of the hardware on which the generated code must run
  - PL/I default – is ARCH(6)
    - produces code that will run on old z990 machines
  - C/C++ default – is ARCH(7)*
    - produces code that will run on z9 machines
- If you specify ARCH(n) and run the generated code on an ARCH(n-1) machine, you will most likely get an operation exception
- So you must set ARCH to the lowest level machine where your generated code will run

*ARCH(7) is the new default in z/OS XL C/C++ V2R1. Default architecture is ARCH(5) for all version before this. ARCH(5) produces code that runs on z900 and older

ARCHitecture -timeline



OPTIMIZE

- Without optimization your code, compiler generate code literally the way you wrote it

- Advantages of NOOPT: very little :–)
  - compile time saving
  - easier to match the code generated with source code written, for the purpose of debugging problems

- Advantages of OPT(2) or higher: a lot ☺
  - Faster code (inlining, constant propagation, more register operation, common sub-expression elimination, dead code elimination, dead store elimination, …)

OPTIMIZE – option

- The PL/I and C++ compilers use the same optimizing backend, but there are differences in what OPT suboptions they support and what they mean:

- PL/I's OPT(2) is a crippled version of C's OPT(2)
  - It helps compile-time be reasonable for large programs
  - But it does produce less than optimal code

- PL/I's OPT(3) is the same as C's OPT(2)
  - It can use a lot of CPU and REGION
  - But it will produce very good and safe code

- C/C++ OPT(2)
  - Inlining, loop unrolling, common sub-expression elimination

- C/C++ has an OPT(3) that is even more aggressive

OPTIMIZE – OPT(3) [C/C++]

- Aggressive code motion, scheduling on computations that have the potential to raise an exception, hardware idiom recognition, instruction level parallelism

- Conformance to IEEE rules are relaxed Floating-point expressions may be rewritten, example fused multiply and add
  - Use –qSTRICT option + O3 to stay conformed to IEEE rules

INLINE - option

- The inline option instructs the compiler to place the code for selected function at the point of call; this is called *inlining*. It eliminates the linkage overhead and exposes the entire inlined function for further optimization.
  - [C/C++] INLINE
  - [PL/I] DFT(INLINE)

INLINE – C example

```
int a, b;
void foo() {
    a = 1;
    b = 2;
    a = bar(a);
    b++;  // no code
}         // motion
```

```
int a, b;
void foo() {
    a = 1;
    b = 2;
    a = a*a;
    b++;
}
```

```
int a, b;
void foo() {
    a = 1;
    a = a*a;
    b = 2;
    b++;
}
```

```
int bar() { return a*a; }
```

INLINE - facts

- This is most beneficial when the inlined function is not large.
- Too much inlining can increase the size of the program.
- [C/C++] You can control
  - how much to inline

    INLINE(AUTO, REPORT, *threshold*, *limit*)
  - when to inline
    - #pragma inline
    - The __attribute__((always_inline))*

*__attribute__((always_inline)) new feature in z/OS XL C/C++ V2R1

INLINE – C/C++ facts

- INLINE(AUTO, REPORT, *threshold*, *limit*)

The maximum size of a function to inline. This is to prevent large functions to be inlined. This is a relative number.

The default is 100.

The maximum size of a function can grow before auto inlining stops. This is to prevent a function to become too large after inlining.

#pragma inline(bar)

```
Int foo()
{
   …
   //x=bar(2);
   x = a*a;
   …
}
```

```
int bar(int a)
{
  return a*a;
}
```

foo grows after inlining

UNROLL - options

- [PL/I] UNROLL(AUTO|NO)
  - [Default] AUTO
    - Compiler permitted to unroll
  - NO
    - Compiler is not permitted to unroll

- [C/C++] UNROLL(AUTO|NO|YES)
  - [Default] Auto
    - Compiler via heuristics and/or honors loops identified via #pragma unroll(nounroll|unroll[(number)])
  - NO
    - Means that the compiler is not permitted to unroll loops in the compilation unit, unless **unroll** or **unroll(n)** pragmas are specified for particular loops.
  - YES
    - Allows the compiler to unroll loops that are annotated (for example, using a pragma), unless it is overridden by **#pragma nounroll**.

UNROLL – C example

```
for (i=0; i<10; ++i)
      sum += i;
```

xlc -qunroll(2) a.c  **all loops in a.c will be unrolled by a factor of 2**

```
#pragma unroll(2)
For (i=0; i<10; ++i)
   sum += i;
```

>xlc a.c  **only this loop will be unrolled by a factor of 2**

Unroll by a factor of 2

```
i=0;
lab2:
if (i>=5) goto lab1;
sum += i;
++i;
sum += i;
++i;
goto lab2;
lab1:
```

FLOAT(AFP)

- Additional floating point, fp8-fp15, registers were added to G5 machine
- These registers are non-volatile, have to be preserved across a call
- More registers provide opportunity to do more operations fast/in register

HGPR - option

- [NO]HGPR(NOPRESERVE|PRESERVE)
  - [PL/I] NOHGPR should be used for all CICS application
  - Gives 16 more working registers to the application
  - PRESERVE will instruct the compiler to store and re-store the high half across the call
  - Defualt is NOHGPR(NOPRESERVE)
  - MetalC is an exception
    - Default is HGPR(PRESERVE)

REDUCE – PL/I option

- The REDUCE option specifies that the compiler is permitted to reduce an assignment of a null string to a structure into simpler operations - even if that means padding bytes might be overwritten.

```
dcl 1 sample ext,
    5 field10 bin fixed(31),
    5 field11 bin fixed(15),
    5 field12 bit(8),
    5 field13 bin fixed(31);

    sample = '';
    w/ REDUCE only one operation to initialize sample to null
    w/ NOREDUCE 4 operations to initialize and padding is unchanged
```

Other PL/I options

- LAXCTL | NOLAXCTL
  - Specifying LAXCTL allows a CONTROLLED variable to be declared with a constant extent and yet to be allocated with a differing extent. NOLAXCTL requires that if a CONTROLLED variable is to be allocated with a varying extent, then that extent must be specified as an asterisk or as a non-constant expression. The following code is illegal under NOLAXCTL:

- RESEXP|NORESEXP
  - Under the NORESEXP compiler option, the compiler will still evaluate all restricted expression occurring in declarations, including those in INITIAL value clauses.
  - For example, under the NORESEXP option, the compiler would not flag the following statement (and the ZERODIVIDE exception would be raised at run time)

Other PL/I options

- LAXCTL | NOLAXCTL
  - Specifying LAXCTL allows a CONTROLLED variable to be declared with a constant extent and yet to be allocated with a differing extent. NOLAXCTL requires that if a CONTROLLED variable is to be allocated with a varying extent, then that extent must be specified as an asterisk or as a non-constant expression. The following code is illegal under NOLAXCTL:

- RESEXP|NORESEXP
  - Under the NORESEXP compiler option, the compiler will still evaluate all restricted expression occurring in declarations, including those in INITIAL value clauses.
  - For example, under the NORESEXP option, the compiler would not flag the following statement (and the ZERODIVIDE exception would be raised at run time)

PL/I - options

- DEFAULT(REORDER NOOVERLAP CONNECTED)
  - Specifying REORDER allows more optimization of your code. Default is REORDER.
  - NOOVERLAP will produce code that performs better; however, if you use NOOVERLAP, you must insure that the source and target never overlap. NOOVERLAP is the default
  - Set the default for whether parameters are connected or nonconnected. CONNECTED allows the parameter to be used as a target or source in record-oriented I/O or as a base in string overlay defining. NONCONNECTED is the default.

XPLINK

- A modern linkage convention that is 2.5 times more efficient than the conventional linkage convention
- We have seen some programs improve by 30%
- You cannot statically link non-XPLINK with XPLINK
- You can call non-XPLINK DLLs from XPLINK DLLs and vice-versa but you must tell the compiler about this so that it can insure the (expensive) switching code gets executed
- If your application contains few switches (as is true of the PL/I compiler where the frontend is not XPLINK and the backend is), then mixing will be beneficial; otherwise it may be very costly

COMPACT

- You want smaller executable? With COMPACT you choose optimization that doesn't impact the size of the final executable or tone down those that they do

STRICT_INDUCTION

- Loop induction variable optimizations can change the result of a program if truncation or sign extension of a loop induction variable occurs as a result of variable overflow or wrap-around.
- The STRICT_INDUCTION option only affects loops which have an induction (loop counter) variable declared as a different size than a register. Unless you intend such variables to overflow or wrap-around, use NOSTRICT_INDUCTION.

Profile Directed Feedback - PDF

- The idea is to use results from sample executions to improve optimization near conditional branches and surrounding infrequently executed code sections. Critical paths are identified and the information is used to guide the optimization.
- This is done in three steps:
  - Build the program with the PDF1 option to instrument the code to collect profiling information.
  - If you are using an MVS™ data set for your PDF file, pre-allocate the PDF data set using RECFM = U and LRECL = 0.
  - Run the program on typical inputs. This is called the *training run*.
  - Build the program again using the PDF2 option.

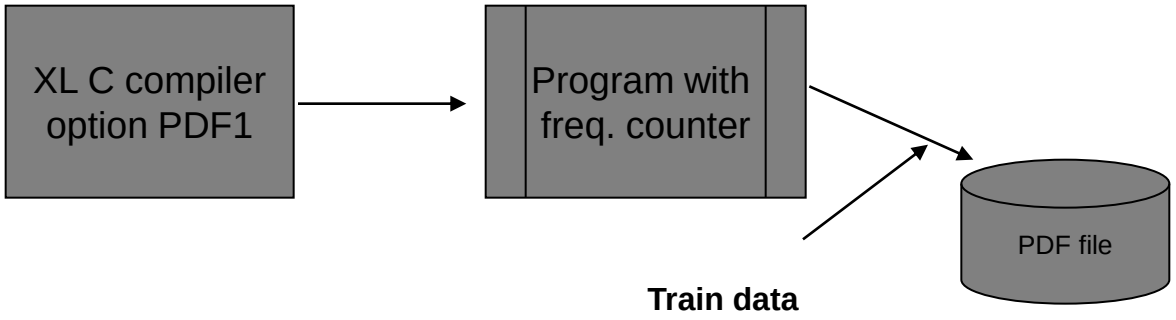
PDF - details

```
a++;
if ( a > 100 ) {   // error
  // code to handle the error
  // could be a large piece of code
        …
}
sum = sum + a;
```

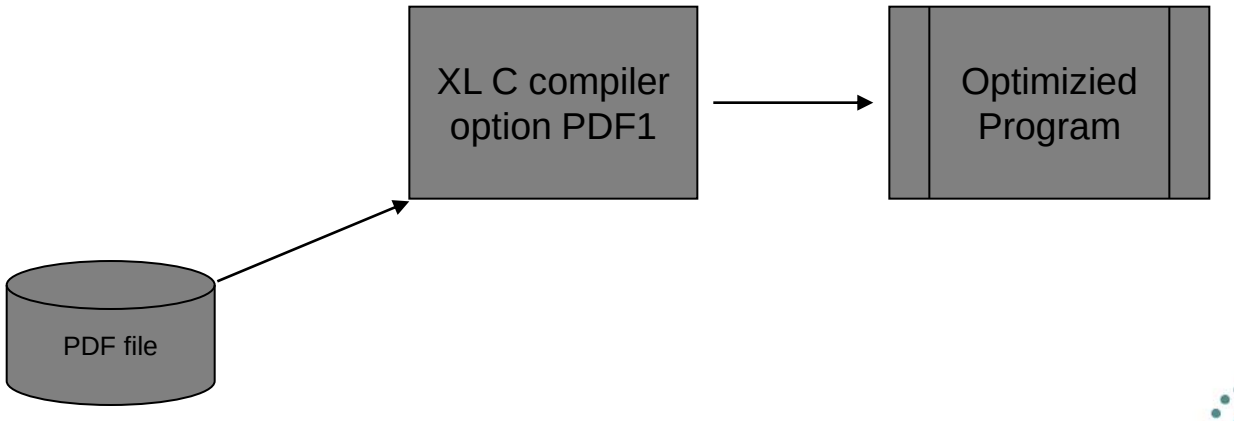
```
a++;
if (a <= 100) goto lab1;
  __count_1++;
 //handle error

  lab1:
 sum = sum + a;
```

PDF - details

1. PDF1



2. PDF2



PDF - facts

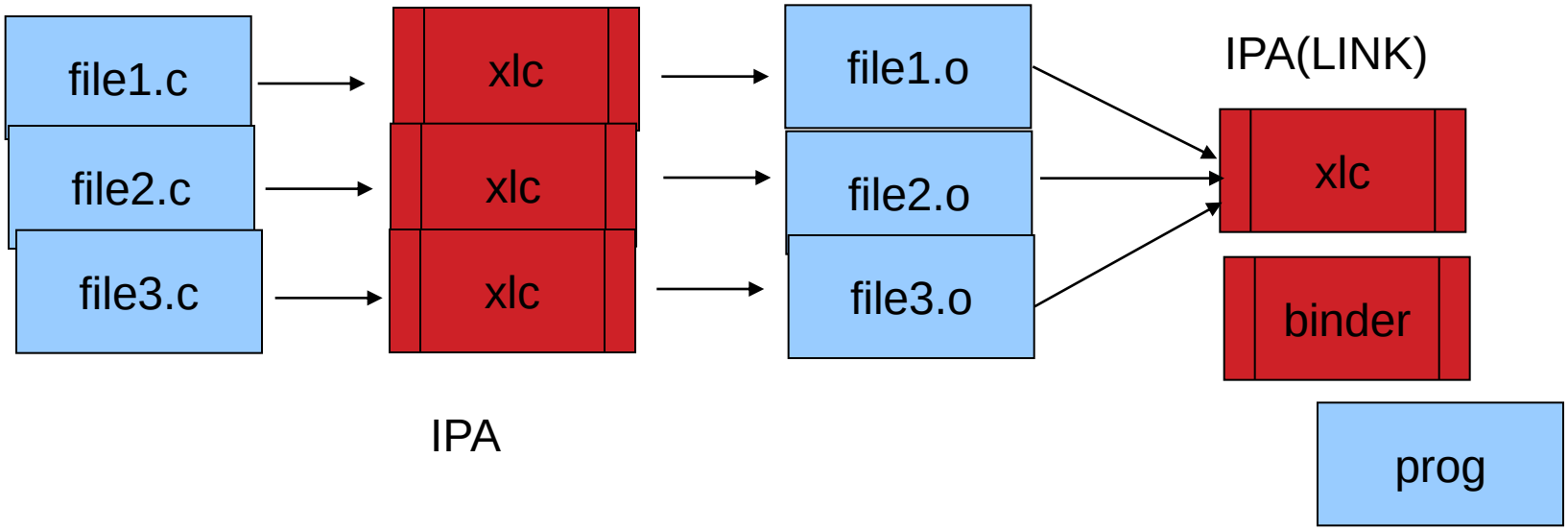
- Use the same: source, options, compiler release for both step one and two

- If you modify the source files, compiler options, or both that are used in step 1, you might see a list of warnings and the benefits from PDF might not apply for the changes from step 1.

- During the PDF2 phase, the compiler issues an information message with a number in the range of 0 - 100. If you have not changed your program between the PDF1 and PDF2 phases, the number is 100, which means that all the profile data can be used to optimize the program. Otherwise, the number is less than 100. If the number is 0, it means that the profile data is completely outdated, and the compiler cannot take advantage of any information. Then you must recompile your program with the PDF1 option and regenerate the profile data.

Inter Procedural Analysis - IPA

- Looks at all the files of a program and optimize globally. The main benefit is that it can do inlining across source files. Further optimization is then possible



IPA - facts

- Note that the time and space needed in the IPA(LINK) step grow rather quickly with the size of the whole program.
- You can control the level of optimization by the LEVEL suboption, 0, 1 or 2. Try LEVEL(0) first.

Write Good Code

- Attempts to be clever and produce “optimal” code have produced:
  - Code that is unreadable
  - Code that cannot be maintained
  - Code that performs worse than less clever solutions
  - Code that fails!

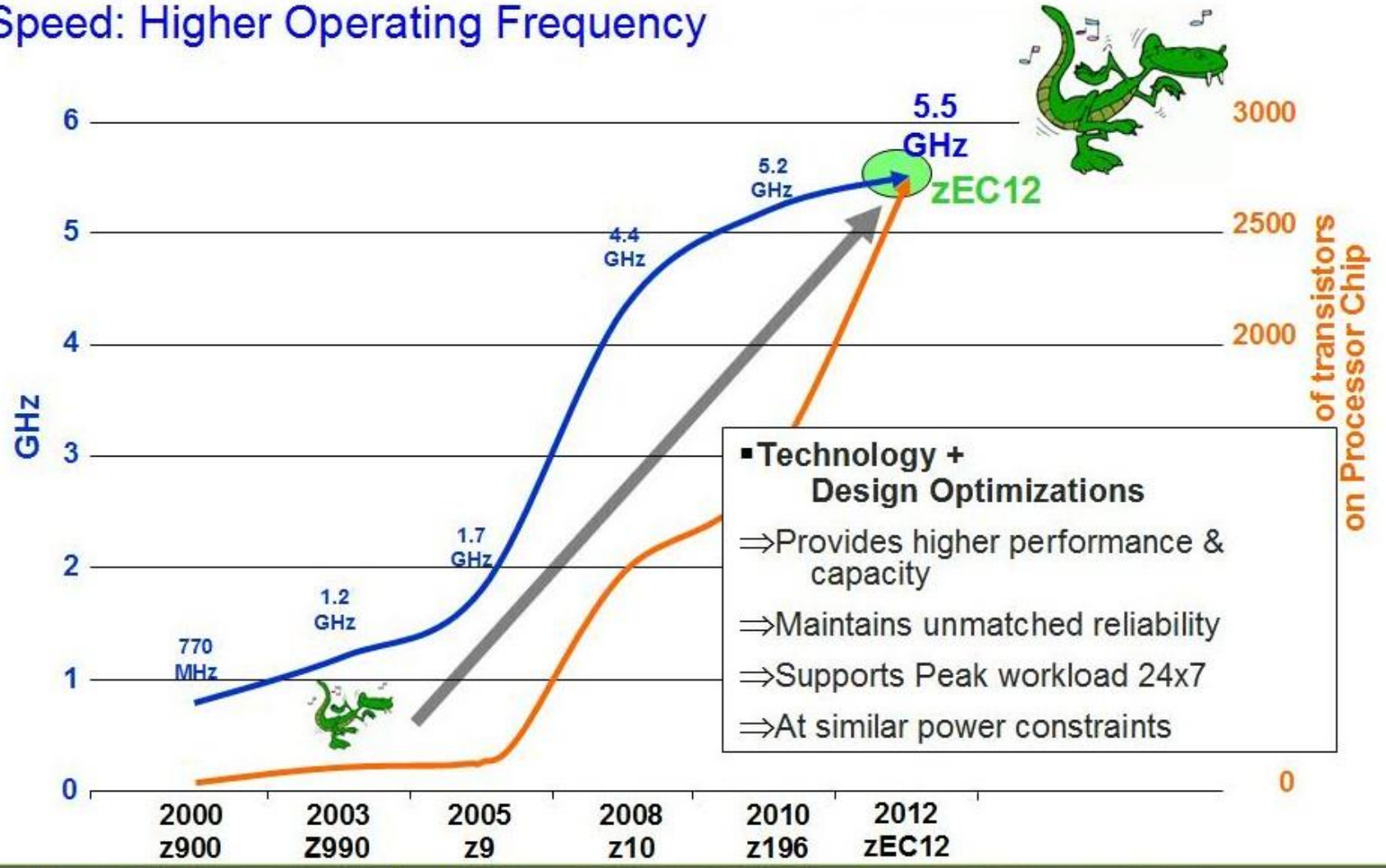
- Readability trumps speed

Install Newer Hardware

- Requires **no**
  - Recompilation
  - Relinking
  - Migration to a new release
  - But can make your code run much faster

- Often the performance boost from moving to new hardware is greater than that from recompiling with the corresponding new ARCH level – However as the Moore's Law no longer holds, we should see less boost with just running on a new hardware

Speed: Higher Operating Frequency



- **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

- **zNext** – OOO+, Architectural Extensions, Enablement for new Software Paradigms

Feedback

- We are collecting feedback for future sessions - which topics are you interested in?
- Looking forward to hearing from you!!
- Please email Peter or Visda

Connect With Us

Cafes

C/C++
http://ibm.com/rational/community/cpp

COBOL
http://ibm.com/rational/community/cobol

Fortran
http://ibm.com/rational/community/fortran

PL/I
http://ibm.com/rational/community/pli

Feature Requests

C/C++
http://ibm.com/developerworks/rfe/?PROD_ID=700

COBOL
http://ibm.com/developerworks/rfe/?PROD_ID=698

Fortran
http://ibm.com/developerworks/rfe/?PROD_ID=701

PL/I
http://ibm.com/developerworks/rfe/?PROD_ID=699

Like IBM Compilers on Facebook

Follow IBM Compilers on Twitter

SHARE
Technology · Connections · Results

Two-Column Slide (Type Size=28)

- Topic A (Type Size=24)
  - Subtopic 1 (Type Size=22)
  - Subtopic 2 (Type Size=22)
  - Subtopic 3 (Type Size=22)
  - Subtopic 4 (Type Size=22)
- Topic B (Type Size=24)

- Topic C (Type Size=24)
  - Subtopic 1 (Type Size=22)
  - Subtopic 2 (Type Size=22)
  - Subtopic 3 (Type Size=22)
    - Sub-subtopic 1 (Type Size=20)
    - Sub-subtopic 2(Type Size=20)
- Topic D (Type Size=24)

Slide with Table

Slide with Text & Graphic