What’s new in z/OS XL C/C++ V1R13 and Enterprise PL/I 4.1

Visda Vokhshoori (visdav@ca.ibm.com)

Peter Elderon (elderon@us.ibm.com)
IBM Corproation
Monday, August, 8, 2011
Session Number 9651

Agenda

- z/OS XL C/C++ V1R13 Highlights
- Enterprise PL/I 4.1 Highlights

z/OS XL C/C++ V1R13 Highlights

- Metal C features
- C++ and C++0X features
- Portability features
- Usability features
- Debugging support features
- Performance improvements

Metal C Features

What is Metal C

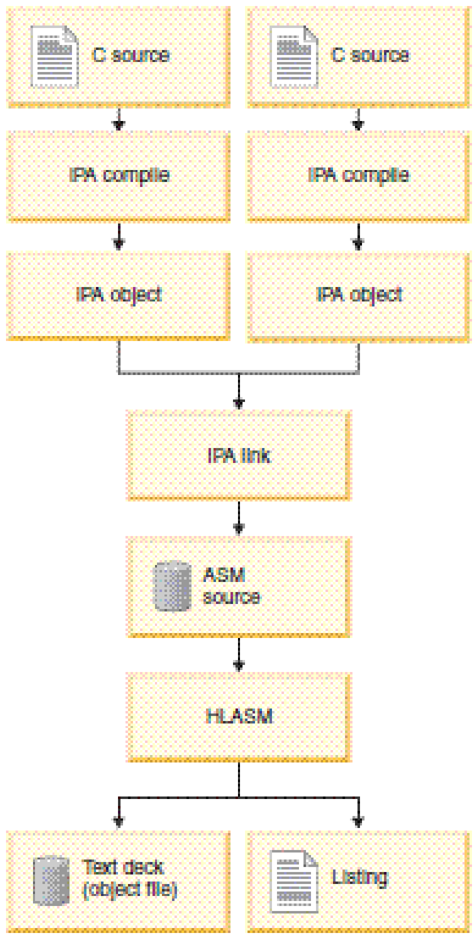
- The XL C compiler generated code requires Language Environment to execute.
- There are requests to run C programs where Language Environment is not available or undesirable.
- The System Programming C facility still require the existence of an environment and there is no provision for user embedded assembler statements.
- The V1R9 XL C compiler introduces a new mode of code generation for system programming purposes.
- We call it Metal C.

Improved Metal C optimization

- To support more advanced optimization for Metal C we enabled IPA and HOT options

- IPA, Inter Procedural Analysis, is a set of compiler optimizations that analyzes and optimizes the program as a whole to improve its performance

- HOT, High Order Transformation, performs loop analysis to reduce their execution time



Building Metal C programs with IPA in USS

IPA Compile

```
xlc –qmetal –c –qipa a.c
xlc –qmetal –c –qipa b.c
```

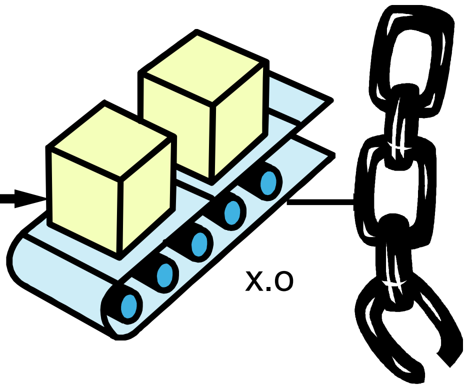
a.o

b.o

IPA Link

```
xlc –qmetal –S –o x.s a.o
b.o
```

x.s



- HLASM output is **NOT** generated
- a.o and b.o contain IPA object only, IPA(OBJECT) is NOT allowed
- LONGNAME option is implicitly turned on

- The output file from IPA Link is the <u>one</u> HLASM source for the whole program
- There could be multiple HLASM structures, one for each partition
- The output file's default name is a.s and is in the cwd

PS- For Building Metal C Programs with IPA using JCL refer to the Back-up slides

Building Metal C Programs with HOT in USS

```
xlc –qMETAL –S –qHOT –o a.s f.c

as a.s

ld –o a.out a.o
```

Constraints

- Mixed addressing mode
- Compiler option:
  - DEBUG
  - REPORT
- IPA sub-options:
  - ATTRIBUTE
  - GONUM
  - PDF sub-options
- IPA control file directives
  - EXPORT
  - NOEXPORTS

New option: DSAUSER

DSAUSER | NODSAUSER

Purpose: To allow users reserve space on the stack; it is addressable in the user prolog code

Default:
NODSAUSER

Usage:
```
/*Prolog code*/
&CCN_DSAUSER SETC ‘#USER_2-@@AUTO@2’
/*Initialize the field*/
STG 0,&CCN_DSAUSER.(,13)
```

Argument parsing

```
/*errorCounter.c*/
int main(int argc, char* argv[])
   {
    if (argc < 2) {
          return 0;
     }
     int errorCount = 0;
     for (int i = 1; i < argc;
   i++) {
          if (argv[i][0] == 'E') {
               errorCount++;
          }
     }
     return errorCount;
}
```

- With V1R13 Metal C supports the parsing of arguments which is a standard C behavior
- This example program executes properly in either USS or batch mode when compiled with V1R13 compiler
- If you want to disable this you can set &CCN_APARSE set symbol to 0 in your prolog code

Function property block

- The Metal C code adds per-function property data that can be used to identify the C function and the associated properties by code scanning or dump reading. This data is called Function Property Block(FPB) and can be found via the new Function Entry Point Marker placed immediately before each function's entry point.
- This feature enables Metal C users to find additional information about a function in a final binary code.
- When the -qCOMPRESS compile option is in effect the function name fields will not be present in the FPB

C++ and C++0X features

New option: TEMPLATEDEPTH

TEMPLATEDEPTH

Purpose: To allow the user to specify their own value for how deep they want the compiler to instantiate recursive template specializations.

Default:

TEMPLATEDEPTH(300)

Usage:

Setting this option to a high value can potentially cause an out-of-memory error because of the complexity and amount of code generated.

TEMPLATEDEPTH

- Before this feature, this code would get
  ```
  CCN5701 (S) The limit on
  nested template
  instantiations has been
  exceeded while instantiating
  "void f<100>()"
  ```
- With V1R13 compiler and option `-qtemplatedepth=400` the program will compile successfully

```
template <int n> void f() {
   f<n-1>();
}

template <> void f<0>() {}
int main() {
  f<400>();
}
```

Temporary lifetime extensions

- This feature is used to extend the lifetime of C++ temporaries beyond that specified by the C++ language standard in 12.2 [class.temporary].
- When enabled, the lifetime of temporaries shall be treated as local variables declared in the inner-most containing lexical scope where possible.
- This feature helps when
  - a user porting an application from another compiler, which may implement late temporary destruction, desires to extend the lifetime of such temporaries in order to replicate the previous non-standard compliant behavior.
  - when a program incorrectly depends on resources, which may have been previously released this feature might help.

Temporary lifetime extensions Example

```
#include<cstdio>
 struct S {
   S() { printf("S::S() ctor at
  0x%lx.\n", this); }
   S(const S& from) {
  printf("S::S(const S&) copy ctor
  at 0x%lx.\n", this); }
   ~S() { printf("S::~S() dtor at
  0x%lx.\n", this); }
 } s1;
 void f(S s) { }
 int main() {
   f(s1);
   printf("hello world.\n");
   return 0;
 }
```

- With

-qlanglvl=tempsaslocals a temporary 's' created for function argument is destroyed after the lexical block of main. By default 's' is destroyed upon returning from 'f'

Temporary lifetime extensions Example

```
#include<cstdio>
 struct S {
   S() { printf"S::S() ctor at
  0x%lx.\n", this); }
   S(const S& from) {
  printf("S::S(const S&) copy ctor
  at 0x%lx.\n", this); }
   ~S() { printf("S::~S() dtor at
  0x%lx.\n", this); }
 } s1;
 void f(S s) { }
 int main() {
   f(s1);
   printf("hello world.\n");
   return 0;
 }
```

```
xlC -qlanglvl=tempsaslocals
-o a.out tempLife.cpp

a.out
S::S() ctor at 0x251208b8.
S::S(const S&) copy ctor at
    0x251252b8.
hello world.
S::~S() dtor at 0x251252b8.
S::~S() dtor at 0x251208b8.
```

rvalue bindings to a non-const reference

- Allow a non-const reference to bind to an rvalue only in the declaration of a function parameter or function return type where an initializer is not required and only for user-defined types.

- Non-compliant compilers may allow a non-const reference to be bound to an rvalue.

- This feature permits also an rvalue to bind to a const-volatile reference and it only applies to top-level CV qualifiers on reference types. The option -qinfo=por will enable an informational message indicating that this binding has taken place despite being illegal.

rvalue bindings to a non-const reference Example

```
struct hey{};
void func(hey& x){}
int main(void)
{
func(hey());
return 0;
}
```

- By default this will be rejected with error:

```
CCN5295 (S) A parameter of type "hey &" cannot be initialized
  with an rvalue of type "hey".
```

- With -qlanglvl=compatrvaluebinding it will compile clean.

Intrinsic complex type

- The complex types, float _Complex, double _Complex and long double _Complex are provided by C++ compiler as built-in types, according to ISO/IEC 9899:1999 Standard.
- Programs with built-in complex types can now be compiled with C++.
- No need to convert intrinsic complex types to Complex class template implementation.
- The feature is normally enabled with -qlanglvl=c99complex.

The complex types and both unary operators __real__ and __imag__ can be enabled with qlanglvl=gnu_complex, it superceeds -qlanglvl=c99complex.

Intrinsic complex type Example

```
#include <stdio.h>
#include <complex.h>
int main() {
    float _Complex a, b;
    a= 2.0f + 3.0f * _Complex_I;
    b = 4.0f - 2.0f * _Complex_I;
    a = a + b;
    printf("a = %f + %f * I . \n", __real__(a), __imag__(a));
}
```

- `Compiling with -qlanglvl=gnu_complex produces the following:`

```
a = 6.000000 + 1.000000 * I
```

[C++0X] Trailing return type

- Primary motivation behind Trailing Return Type feature is the ability to declare function templates whose return type depends on the types of the template arguments.

Trailing return type Example

```
template <class A, class B>
auto multiply(A a, B b)->decltype(a*b)
{
   return a*b;
}
```

```
Compiler in R13 -qlanglvl=autotypededuction:decltype
```

Portability features

Suppress warning for text following #else-#endif

- Allows text on the same line after the #endif and #else preprocessing directives
- Code ported from other platforms may have this non-standard extension

This is a deviation from the standard so the code will be less portable to other platforms that do not have this extension.

Function attributes (gnu_inline, used, malloc)

- gnu_inline: Uses pre-C99 GCC inline behaviour
- used: Marks a function as used so it is not removed
- malloc: Any non-null pointer returned cannot alias any other pointer that is valid at the time of the function call. Can help increase runtime performance

- This is a deviation from the standard and hence the code may be not be portable to other compilers.

Function attributes Example

```
extern inline __attribute__((gnu_inline)) f() {…};
static inline __attribute__((gnu_inline))  b() {…};
__attribute__((used)) void f() { }
  int main() { f(); }
void* f()  __attribute__ ((__malloc__)) { ... }
```

Addressable labels

- Add support for the Labels-as-values and Computed-goto features that are implemented by GCC.

- Makes porting code over to our compiler from other ones easier.

- This is a deviation from the standard and hence the code may be not be portable to other compilers.

Addressable labels Example

```
/*mysource.c*/
#define good 0
#define bad 1
int main(void) {
  void* la = &&label1;
  goto *la;
  return bad;
label1:
  return good;
}
```

```
Compile and run:
xlc mysource.c -o ./a.out
a.out
Returns with code 0
```

Usability

New Hardware Built-ins

- Interlocked-storage-access instructions, available on models where the interlocked-access-facility is installed, **provide a means by which a load, update and store operation can be performed with interlocked update in a single instruction.** Supported interlocked-storage-access instructions are:
  - Load and Add (LAA, LAAG)
  - Load and Add Logical (LAAL, LAALG)
  - Load and And (LAN, LANG)
  - Load and Exclusive Or (LAX, LAXG)
  - Load and Or (LAO, LAOG)
  - Load Pair Disjoint (LPD, LPDG)

- This is an offering for z196 hardware, it is implemented under ARCH(9)

HFP Multiply and Add/Subtract Facility

- The complier now can generate fused multiply and add/subtract instructions for hexadecimal floating point calculations.
- This was not allowed in previous releases due to run-time performance reasons.
- With the use of ARCH(9) fused multiply and add/subtract is enabled. This allows potential performance increases for these calculations.

- Using FLOAT(MAF) with FLOAT(HEX) in ARCH(9) should improve the run-time performance of floating point application

Informational messages on by default on USS

V1R12 and older releases

Defaults

➢FLAG(I)

➢For the z/OS UNIX System Services utilities, the default for a regular compile is FLAG(W).

V1R13 and later release

Defaults

➢FLAG(I)

- Any existing compilations that did not turn on FLAG(I) and used the INFO or CHECKOUT may see the additional informational messages.

INFO/CHECKOUT defaults

- Some of the default sub-options for INFO and CHECKOUT have changed to avoid emitting non-problem messages and to make CHECKOUT more similar to INFO.
- Using ALL sub-option will no longer emit pre-processing trace information, *PPT*, by default instead it will only emit the potential non-aliasing error cases, *ALS*.
- This should make the information emitted by default sub-option of CHECKOUT and INFO similar and will aid migration.

- Users using INFO, INFO(ALL) or CHECKOUT will potentially get different messages. This effects the #pragma options as well

Debugging support features

Hookless Debug

- dbx added support for debugging programs compiled without hooks
- Intended to allow you to debug programs whose sizes and performance characteristics are more closely aligned with production programs

Debug information for inline-d procedures

- Ability to set entry breakpoints at all inline-d instances

- The support is invoked by option:
    - DEBUG(FORMAT(DWARF)) + OPT

Performance enhancement features

Performance of C/C++ code on z196

- % improvement = (geometric mean of A)/(geometric mean of B), both running on z196, where:
  - A = programs compiled by V1R13
  - B = programs compiled by V1R12

* This is based on internal IBM lab measurements using the following compiler options:

- For 31-bit: ILP32, XPLINK, HGPR, OPT(3), HOT, IPA(LEVEL(2)), PDF, ARCH(9), TUNE(9)
- For 64-bit: LP64, XPLINK, HGPR, OPT(3), HOT, IPA(LEVEL(2)), PDF, ARCH(9), TUNE(9)

Performance results for specific applications will vary; some factors affecting performance are the source code and the compiler options specified.

Performance of C/C++ code on z196

- Programs compiled with the V1R13 compiler may show significant performance improvement* when compared to the same programs compiled with V1R12
  - 4% improvement was observed on a set of 31-bit CPU intensive integer based programs
  - 7% improvement was observed on a set of 64-bit CPU intensive integer based programs
  - 7% improvement was observed on a set of CPU intensive floating-point based programs

    (31-bit & 64-bit)

Performance of C/C++ code on z196

- For a detailed description of how to improve your application’s performance see also

www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP101796

Enterprise PL/I 4.1 Highlights

- Performance Enhancement Features
- Improved Debug Tool support
- XML validation
- New (sub)options for better quality
- Miscellaneous user requirements

Performance Enhancement Features

REFER

- Code that uses elements of structures with multiple REFERs can be very expensive: each reference uses a costly library call to remap the structure
- Now, for structures where all elements are byte-aligned, those calls will be avoided and straightforward inline code generated
- If all elements are byte-aligned, no padding is possible and thus the address calculations are relatively simple
- To insure all elements are byte-aligned
  - Specify UNALIGNED on the level-1 part of declared
  - Declare any NONVARYING BIT as ALIGNED

REFER

E.g., consider these declares (and note the UNALIGNED):

```
dcl (first,middle,last) char(*) var;

dcl f_len fixed bin(31);
dcl m_len fixed bin(31);
dcl l_len fixed bin(31);
dcl q pointer;
dcl
   1 name based UNALIGNED,
     2 len_first fixed binary(31),
     2 first char( f_len refer(len_first) ),
     2 len_middle fixed binary(31),
     2 middle char( m_len refer(len_middle) ),
     2 len_last fixed binary(31),
     2 last char( l_len refer(len_last) );
```

REFER

- A library call is still made to map the structure for the allocate, but the 6 library calls that would have been done to make the assignments have been eliminated:

```
f_len = length(first);
m_len = length(middle);
l_len = length(last);

allocate name set(q);

q->name.first = first;
q->name.middle = middle;
q->name.last = last;
```

INDEX

- The code generated for the INDEX built-in function has been optimized by Enterprise PL/I when there are only 2 arguments
- The compilers before Enterprise PL/I permitted only 2 arguments, but Enterprise PL/I allows a third argument to specify where the search should start
- This usage has now also been optimized when the second argument is just a single byte, e.g. a semicolon or a blank

INDEX

- This can be very useful in code that processes some text in semicolon delimited chunks, as in:

```
pos = 0;
pos = index( text, ‘;’, pos+1 );
do while( pos > 0 );
    /* process text to semicolon */
   pos = index( text, ‘;’, pos+1 );
end;
```

Improved Debug Tool support

Reduced object size

- DebugTool uses the statement number table generated by the GONUMBER option (which is why TEST generally forces GONUMBER to be on)
- With Enterprise V3, the GONUMBBER table was part of the generated object code (and hence part of the linked load module) even if TEST(SEPARATE) was used
- With Enterprise V4, if you specify TEST(SEP) and GONUMBER(SEP), the compiler will place the statement number table in the side file and thus significantly reduce the size of the generated object
- For compatibility, the default for GN is GN(NOSEP)

Improved automonitor support

- Under Enterprise V3, for its AUTOMONITOR, the compiler generated information that specified only the name of the variable, but omitted any subscripts or pointer qualifications
- Under Enterprise V4, when using TEST(SEPARATE), the compiler generates information that names the fully qualified reference
- E.g. for a statement of the form A(2) = B(2); A(2), and B(2) will be listed in the monitor window (rather than all of A and all of B)

Most support for implicit BASED

- Under Enterprise V3, the compiler generated a symbol table that allowed implicit locator references for variables declared as BASED on simple scalars

- With Enterprise V4, when using TEST(SEPARATE), the compiler will generate information to identify complicated implicit locator references such as those for a variable is BASED on
  - ADDR of array element or
  - Other built-in functions (such as ADDRDATA, POINTERADD, etc)

DCL and XREF information

- Under Enterprise V4, when using TEST(SEPARATE), the compiler will include in the side file, information identifying the source lines for
  - Declares
  - References   (xref refs)
  - Assignments (xref sets)
- This will help enable DebugTool to provide information on these declares and/or to allow you to search for these statements etc.

XML Validation

PLISAXD

- The new PLISAXD built-in subroutine is like PLISAXC except that it will cause the incoming XML to be validated
- It requires an additional argument: an Optimized Schema Representation
- Like PLISAXC, PLISAXD uses the System Services XML Parser
- And its arguments are much like PLISAXC

PLISAXD

- In order, its arguments are
  - An event structure
  - A token passed pack to the event functions
  - The address of a buffer containing the XML
  - The size of the buffer
  - The address of the buffer containing the OSR
  - An optional codepage identifier
- The only difference from PLISAXC is the 5th parameter
- The even structure is the same as PLISAXC

PLISAXD

- While the event structure is the same as for the PLISAXC, the exception event may see some additional exceptions found by the validation
- The z/OS Unix command xsdosrg will generate a file containing the OSR for a given schema
- You must do this before trying to run code using PLISAXD
- And the before invoking PLISAXD, you must read the OSR into a buffer
- The Programming Guide has more details

New (sub)options for Better Quality

DEPRECATE (Racon – MR0427097311)

- The new DEPRECATE option will flag the usage of various include files, built-in functions or variables that you wish to deprecate. It will flag via:
    - the BUILTIN suboption, any specified name declared as a BUILTIN
    - the ENTRY suboption, any specified name declared as a level-1 ENTRY
    - the INCLUDE suboption, any specified name used in an %INCLUDE statement
    - the VARIABLE suboption, any specified name declared as level-1 name and not having the BUILTIN or ENTRY attribute

DEPRECATE (Racon – MR0427097311)

- So if you want to flag the usage of UNSPEC and any variable named just I, J or N, you could specify
  - DEPRECATE (BUILTIN(UNSPEC) VARIABLE(I,J,N))
- Specifying one of the suboptions does not change the setting of any of the other suboptions specified previously. So the above could also be specified as:
  - DEPERCATE(BUILTIN(UNSPEC)) DEPRECATE(VARIABLE(I,J,N))

NOGLOBALDO (Telcordia – MR1104096225)

- Under the new RULES(NOGLOBALDO) option, the compiler will flag any DO statement where the loop control variable is declared in a parent procedure –as in this code

```
a: proc;
   dcl jx fixed bin;
   call b;
   b: proc;
     do jx = 17 to 29;
   end;
  end b;
end a;
```

NOGLOBALDO (Telcordia – MR1104096225)

- This usage creates
  - non-transparent code (it is rarely good when a subroutine changes the value of a variable in a parent procedure)
  - less optimized code
- So flagging it is good
- For compatibility, the default is RULES(GLOBALDO)

NOPADDING (Telcordia –MR1110093235)

- Under the new RULES(NOPADDING) option, the compiler will flag any structure where it can tell that there will be padding bytes

- For compatibility, the default is RULES(PADDING)

- RULES(NOPADDING) would flag, for example

```
dcl
   1 a aligned,
   2 b fixed bin(31),
   2 c char(3),
   2 d fixed bin(31);
```

SHARE in Orlando 2011

Miscellaneous User Requirements

Init of typed structures (Wuestenrot - MR0312104052)

- In particular, the INIT attribute will now be allowed on leaf elements of a DEFINE STRUCTURE statement
- However, INIT CALL, INIT TO, and VALUE will still not be allowed on elements of a DEFINE STRUCTURE statement
- For example, the following is now allowed

```
define struct
   1 b,
     2 b1 fixed bin init(17),
     2 b2 fixed bin init(19);
```

Init of typed structures (Wuestenrot - MR0312104052)

- The new VALUE type-function may then be used to initialize or assign to a variable having the corresponding structure type, e.g.

```
define struct
   1 b,
     2 b1 fixed bin init(17),
     2 b2 fixed bin init(19);
define struct
   1 c,
     2 c1 type b init( value(: b :) ),
     2 c2 fixed bin init(23);
dcl x type c static init( value(: c :) );
dcl y type c; y = value(: c :);
```

Init of typed structures (Wuestenrot - MR0312104052)

- The VALUE function has one mandatory argument that must be the name of a typed structure, and it returns an instance of that typed structure with its initial values
- If the VALUE function is used with a structure type that is only partially initialized, uninitialized bytes and bits will be zeroed out
- The VALUE function may not be used with a structure type containing no elements with the INITIAL attribute

SQL XREF (LVM -MR1112095051)

- The integrated SQL preprocessor will now accept (NO)XREF as an option
- Under XREF, it will produce an XREF listing like that produced by the old SQL precompiler
- This means that the integrated SQL preprocessor provides a full superset of the function available via the SQL precompiler

ONAREA (Telcordia-MR1217095934)

- If AREA has been raised, ONAREA will return a string specifying the AREA reference for which the allocate failed
- So if ALLOCATE X IN(A) fails, ONAREA will return the string ‘‘A”
- And if ALLOCATE X IN(A1.A2(N)) fails, ONAREA will return “A1.A2(N)”

REENTRANT Proc’s (StateFarm - MR102909480)

- Before Enterprise PL/I, specifying REENTRANT in the OPTIONS attribute of a PROC statement changed the code that was generated and was required if the code was supposed to be reentrant

- With Enterprise PL/I, it did nothing

- With 4.1, it will now cause the compile to issue a message unless you use either

  - the RENT option, or
  - the DFT(NONASGN) option

- This is under the assumption that such proc’s are supposed to be reentrant (and in that case, the compiler should flag any assign to static)

VALUE in structures (MR0213091212)

- The VALUE attribute is now allowed in (non-typed) structures, but then
- All leaf elements of the structure must have the VALUE attribute
- The structure must not contain any unions or arrays
- This makes conversion of old declares using STATIC INIT to VALUE easier (and the use of VALUE will let the compiler produce better code)
- It also allows you to have “namespaces” of VALUE

Learn more at:

- IBM Rational software
- IBM Rational Software Delivery Platform
- Process and portfolio management
- Change and release management
- Quality management
- Architecture management
- Rational trial downloads
- developerWorks Rational
- IBM Rational TV
- IBM Rational Partner Community
- IBM Rational C/C++ Cafe
- IBM Rational PL/I Cafe

Back-up slides

Building Metal C Program with IPA with JCL

```
//*IPA Compile
//COMPID1 EXEC EDCC,
// INFILE='VISDAV.METALIPA.SOURCE',
// OUTFILE='VISDAV.METALIPA.OBJECT,DISP=SHR',
// CRUN='',
// CPARM='OPTFILE(DD:XOPTS)'
//XOPTS  DD DATA,DLM='/>'
IPA(NOLINK) LONGNAME NOSEARCH
METAL
/>
//*IPA Link
//IPALIN1 EXEC CBCI,
// IRUN='',
// IPARM='OPTFILE(DD:OPTS)'
//OBJECT DD DSN=VISDAV.METALIP.IPA.OBJECT,DISP=SHR
//SYSIN  DD DSN='VISDAV.METALIPA.OBJECT',DISP=SHR'
//SYSLIN DD DSN='VISDAV.METALIPA.SOURCE',DISP=SHR
//OPTS DD DATA,DLM='/>'
METAL GENASM
/>
```

Performance of C/C++ code on z196

- What if you don't recompile?
  - We compared the performance of the same binaries executing on z196 and z10.Binaries were built using the V1R11 compiler.

- On z196 we achieved overall improvements of:50% for a set of cpu intensive integer based programs*.
  - 125% for a set of cpu intensive floating point based programs*.
  - This is based on internal IBM lab measurements using the following compiler options:
  - ILP32, XPLINK, HGPR, OPT(3), HOT, IPA(LEVEL(2), PDF, ARCH(8), TUNE(8)
  - Performance results for specific applications will vary; some factors affecting performance are the source
- code and the compiler options specified.