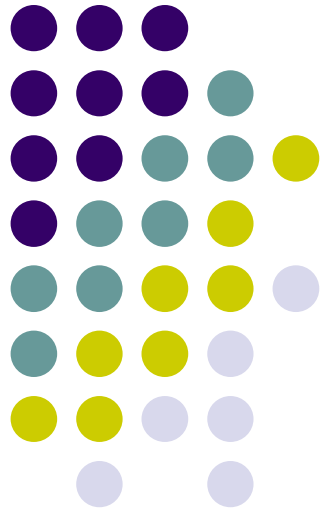
Converting your Language Environment® C/C++ Applications to XPLink for 64ZSP04587USE

Corey Bryant
IBM Poughkeepsie
bryntcor@us.ibm.com



Trademarks

The following are trademarks of the International Business Machines Corporation in the United States and/or other countries.

- IBM*
- z/OS*
- OS/390*
- Language Environment*
- CICS*
- DB2*
- MVS
- IMS
- Redbooks

* Registered trademarks of IBM Corporation

The following are trademarks or registered trademarks of other companies.

Java and all Java-related trademarks and logos are trademarks of Sun Microsystems, Inc., in the United States and other countries.
Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
Microsoft, Windows and Windows NT are registered trademarks of Microsoft Corporation.
UNIX is a registered trademark of The Open Group in the United States and other countries.
SET and Secure Electronic Transaction are trademarks owned by SET Secure Electronic Transaction LLC.

* All other products may be trademarks or registered trademarks of their respective companies.

Notes:

Performance is in Internal Throughput Rate (ITR) ratio based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput improvements equivalent to the performance ratios stated here.

IBM hardware products are manufactured from new parts, or new and serviceable used parts. Regardless, our warranty terms apply.

All customer examples cited or described in this presentation are presented as illustrations of the manner in which some customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics will vary depending on individual customer configurations and conditions.

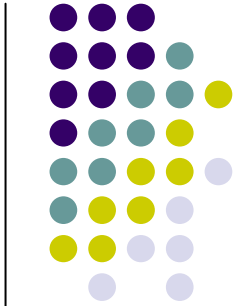
This publication was produced in the United States. IBM may not offer the products, services or features discussed in this document in other countries, and the information may be subject to change without notice. Consult your local IBM business contact for information on the product or services available in your area.

All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

Information about non-IBM products is obtained from the manufacturers of those products or their published announcements. IBM has not tested those products and cannot confirm the performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Prices subject to change without notice. Contact your IBM representative or Business Partner for the most current pricing in your geography.

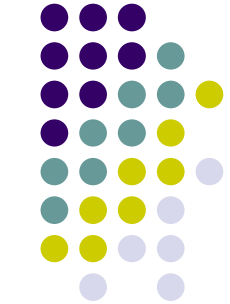
Contents



Background

- New workloads on OS/390® and z/OS® consist largely of applications written on other platforms where function calls were "free“.
- Some applications measure ~25% of execution time spent in function call overhead!
  - These are the most serious in Object-Oriented applications, where functions tend to be smaller. That is, a higher ratio of functions calls to lines of "application code“.
- New, especially ported, workloads are primarily all C or C++, with little or no COBOL or Assembler.
  - This is the real target audience for XPLink.

XPLink Overview

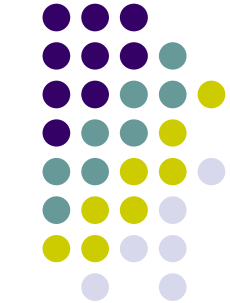


The objective of XPLink is to provide, for a specific type of application:

- Improved call linkage performance (up to 50% reduction in linkage instructions)
- Reduced function footprint in memory
- A common linkage for C/C++ (and DLLs)
- Compatibility with existing (non-XPLink) code
- No effect on existing applications

These are the characteristics expected to be exploited by 64-bit applications. Thus 64-bit is strictly XPLink!

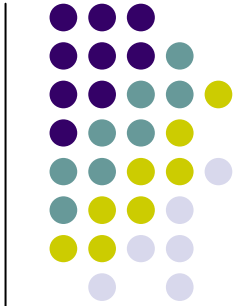
XPLink Overview



What's Happening with Today's Linkage?

- It will probably be with us "forever".
  - At least in the existing environment
  - But perhaps not in future environments that are incompatible for other reasons (64-bit, for example, is completely XPLink)
- There is compatibility support between old and new linkages across Program Object (DLL call) boundaries.
  - At some cost in performance
  - For 31-bit only (64-bit is completely XPLink)

XPLink Overview



Did you say “improved call linkage performance”?

- YES! (Well for specific applications.) Following is a function prolog linkage comparison (the instructions that set up a new stack frame):

"Old" 31-bit f2() prolog

```
000000  47F0  F022        B     34(,r15)
000004  01C3C5C5                CEE eyecatcher
000008  00000098                DSA size
00000C  000000C0                =A(PPA1-f1)
000010  ... stack extension path
000022  90E4  D00C        STM   r14,r4,12(r13)
000026  58E0  D04C        L     r14,76(,r13)
00002A  4100  E098        LA    r0,152(,r14)
00002E  5500  C314        CL    r0,788(,r12)
000032  4130  F03A        LA    r3,58(,r15)
000036  4720  F014        BH    20(,r15)
00003A  58F0  C280        L     r15,640(,r12)
00003E  90F0  E048        STM   r15,r0,72(r14)
000042  9210  E000        MVI   0(r14),16
000046  50D0  E004        ST    r13,4(,r14)
00004A  18DE              LR    r13,r14
```

```
void f1(void) {
   f2();
   NOP flag
};

void f2(void) {
   // ...
};
```

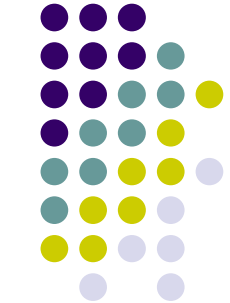
XPLink 31-bit f2() prolog

```
000000  9057  4784        STM   r5,r7,1924(r4)
000004  A74A  FF80        AHI   r4,H'-128'
```

XPLink 64-bit f2() prolog

```
000000  EB57  4708 0024   STMG  r5,r7,1800(r4)
000006  A74B  FF00        AGHI  r4,H'-256'
```

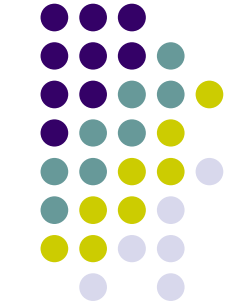
XPLink Major Differences



New calling convention

- Registers 13, 14, and 15 are just work registers
- R6 and R7 are used for linkage (ie. BASR 7,6)
  - formerly R14 and R15 were used for linkage (ie. BALR 14,15)
- R5 contains called function's own portion of WSA (its environment)
  - formerly R0 contained address of WSA, the called function had to compute the address of its own piece of Writeable Static
- No base register (R6) assumed on entry, call *may* have been made via Relative Branch
- Return register (R7) not preserved, caller cannot rely on it on return

XPLink Major Differences



Improved Parameter / Return Value Passing

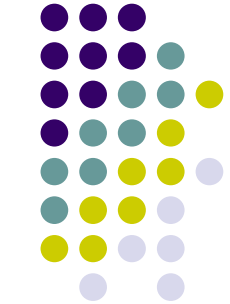
- Arguments passed via fixed location in caller's stack
- Arguments are directly addressable by called function
  - Also addressable by caller using same base reg as its own auto storage
- First 3 argument words passed in GPRs 1-3 (31-bit)
- First 3 argument doublewords passed in GPRs 1-3 (64-bit)
- Up to 4 floating point arguments passed in FPRs
- Remaining arguments passed in storage
- Return value is in GPR 3 (extended value in R2 and R1)

XPLink Major Differences

New Stack Layout

- Grows towards lower addresses
  - Called function knows how to adjust stack pointer; it doesn't have to be passed from caller (LE's NAB)
  - Beware of stack overruns with downward growing stack!
    - For example, buffer overflow beyond array's bounds extends automatic storage and overwrites calling function's stack frame.

XPLink Major Differences



New Stack Overflow Detection

- No explicit test (well.. usually) for stack overflow
  - Based on guard page size
  - Functions with large stack frames still need explicit test
    - **In 64-bit large is immense**
- Called function stores into new stack frame, storage protection used to determine if stack needs extending
  - Prolog consists of updating stack pointer and saving registers there - registers are stored in called function's stack frame, not caller's

XPLink Major Differences

Where Have all the Instructions Gone?

"Old" 31-bit f2() prolog

```
0000  47F0  F022          B      34(,r15)
0004  01C3C5C5                   CEE eyecatcher
0008  00000098                   DSA size
000C  000000C0                   =A(PPA1-f1)
0010  ... stack extension path
0022  90E4  D00C          STM    r14,r4,12(r13)
0026  58E0  D04C          L      r14,76(,r13)
002A  4100  E098          LA     r0,152(,r14)
002E  5500  C314          CL     r0,788(,r12)
0032  4130  F03A          LA     r3,58(,r15)
0036  4720  F014          BH     20(,r15)
003A  58F0  C280          L      r15,640(,r12)
003E  90F0  E048          STM    r15,r0,72(r14)
0042  9210  E000          MVI    0(r14),16
0046  50D0  E004          ST     r13,4(,r14)
004A  18DE                LR     r13,r14
```

moving control information out of line

better register conventions reduce number of registers saved (7 vs 3 here)

detecting overflow with guard page

downward-growing stack allows these 3 instructions to be replaced with a single AHI

XPLink 31-bit f2() prolog

```
0000  9057  4784          STM    r5,r7,1924(r4)
0004  A74A  FF80          AHI    r4,H'-128'
```

biased stack pointer allows this instead of:

```
          AHI    r4,H'-128'
*         wait for register 4 to be available
          STM    r6,r7,8(r4)
```

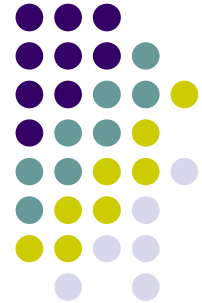
other improvements:

- no base register
- no Library Work Area
- no stack marking

static instead of dynamic stack information (a compiler option can be used to force the saving of the backchain by increasing the range of the initial STM instruction)

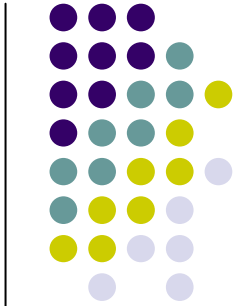
```
void f1(void) {
   f2();
   NOP flag
};

void f2(void) {
   // ...
};
```

XPLink Register Conventions



Register	XPLink Usage on Entry	XPLink Usage on Return
GPR 0	undefined	not preserved
GPR 1	1st word of argument list or undefined	not preserved or 1st word of a returned aggregate
GPR 2	2nd word of argument list or undefined	not preserved or 2nd word of returned aggregate or high half of 64-bit integer return value
GPR 3	3rd word of argument list or undefined	not preserved or 3rd word of returned aggregate or 31-bit return value
GPR 4	Address of Stack Frame minus 2048 bytes	preserved
GPR 5	Address of called function's environment or, for internal functions, containing scope's stack frame	not preserved
GPR 6	Entry point address or undefined	not preserved
GPR 7	Return address	not preserved
GPR 8-11	undefined	preserved
GPR 12	Undefined; Or for 31-bit LE-conforming applications: Pointer to (thread-specific) CAA - must be set on entry to any function	preserved (in either case)
GPR 13-15	undefined	preserved

XPLink Register Conventions

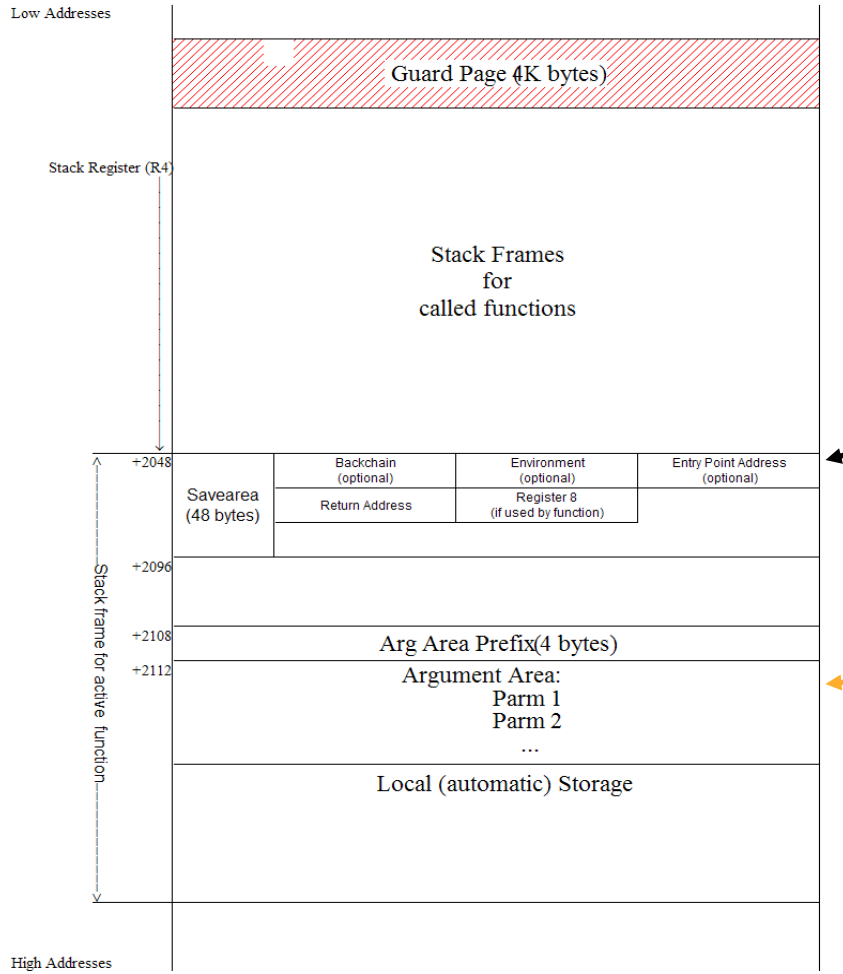


Comparing “Old” vs. XPLink Register Conventions:

Register Usage	“Old”	XPLink
Stack Ptr	R13	R4 (biased)
Return Addr	R14	R7
Entry pt on entry	R15	R6 (unless called by branch relative)
Environment	R0 (WSA)	R5
CAA Address	R12	- R12 in 31-bit - from LAA in 64-bit
Input Parm List	Address in R1	In caller's DSA, first 3 ints in R1, R2, R3, float pt values in FPR0, 2, 4, 6
Return Code/value	R15	R3 (extended value in R2 and R1)
Start of callee's stack frame	Caller's NAB value	Caller's R4 minus callee's stack frame size

XPLink Function Call Example

Following the XPLink stack during a function call.

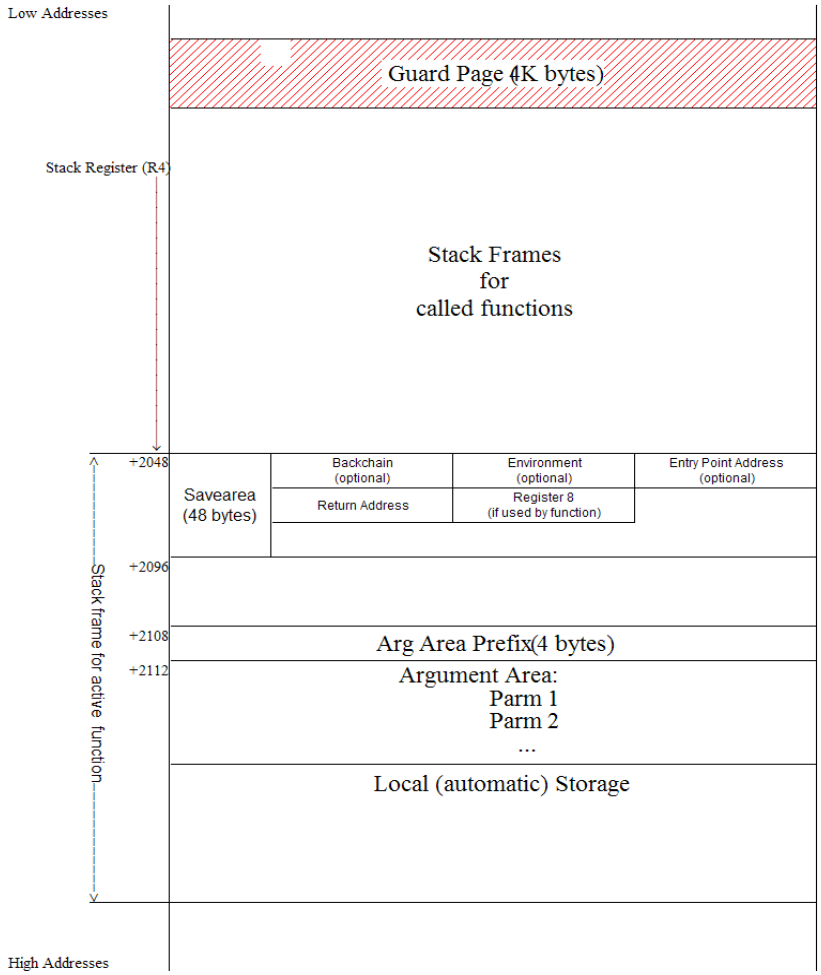



```
void f1(void) {
   f2();
   nop  flag
};

void f2(void) {
   // ...
   //
   // ...
};
```

1 **Construct arguments in current stack frame's argument area**

some arguments are passed in registers; more about this later

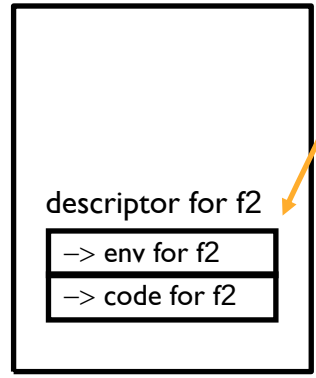
XPLink Function Call Example




```
void f1(void) {
   f2();
   nop  flag
};

void f2(void) {
   // ...
   //
   // ...
};
```

```
000050  9856  5000      LM    r5,r6,=A(f2)(r5,0)
000054  0D76             BASR  r7,r6
000056  4700  FFFD       NOP   4093(,r15)
```

Environment for f2 in WSA

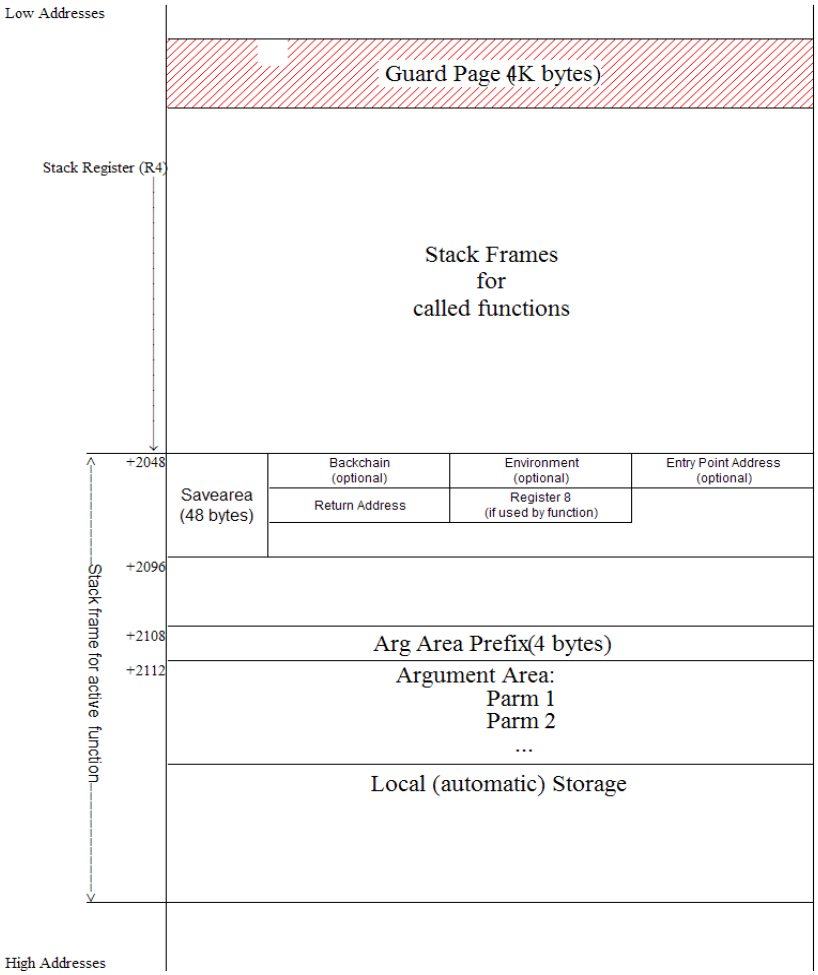


1 Construct arguments in current stack frame's argument area

2 **Pick up address of function's "environment" and code from the function descriptor**

the environment is an area of Writeable Static associated with the called function

XPLink Function Call Example



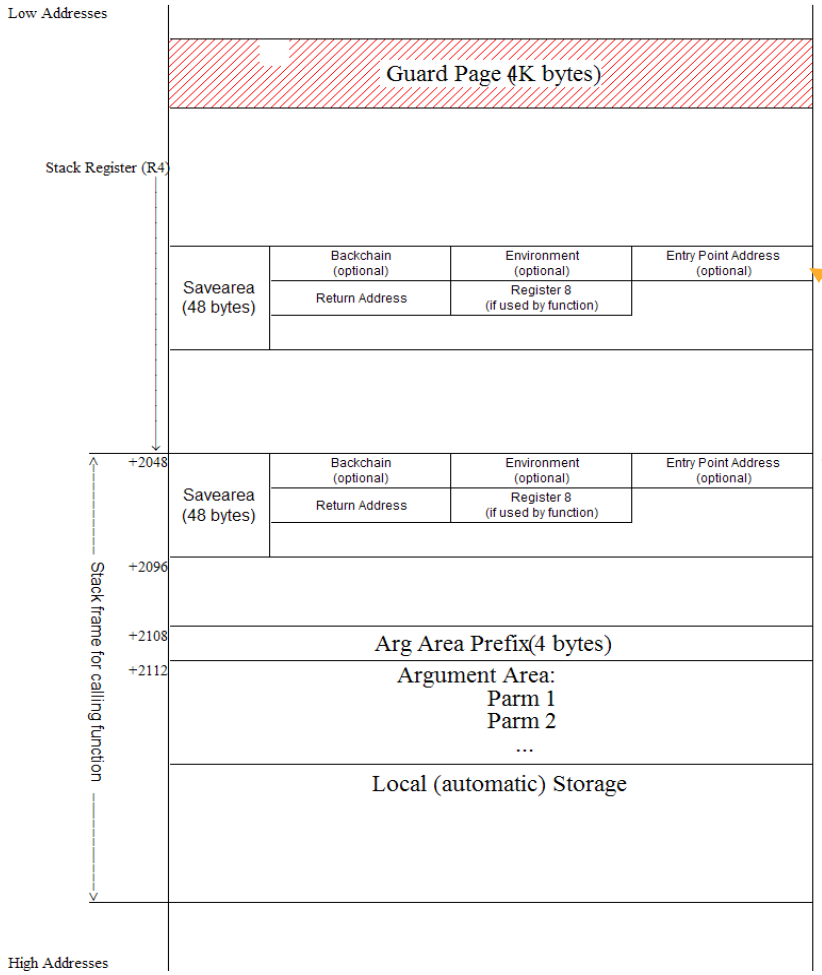

```
void f1(void) {
  f2();
  nop  flag
};

void f2(void) {
  // ...
  //
  // ...
};
```

```
000050  9856  5000      LM    r5,r6,=A(f2)(r5,0)
000054  0D76            BASR  r7,r6
000056  4700  FFFD      NOP   4093(,r15)
```

1. Construct arguments in current stack frame's argument area
2. Pick up addresses of function's "environment" and code from the function descriptor
3. **Link (BASR or BRAS) to called function**

if the called function is in a DLL, the LM will pick up a handle for the called function and the address of the DLL loader, as initialized by the Binder on detecting that the called function is imported

XPLink Function Call Example



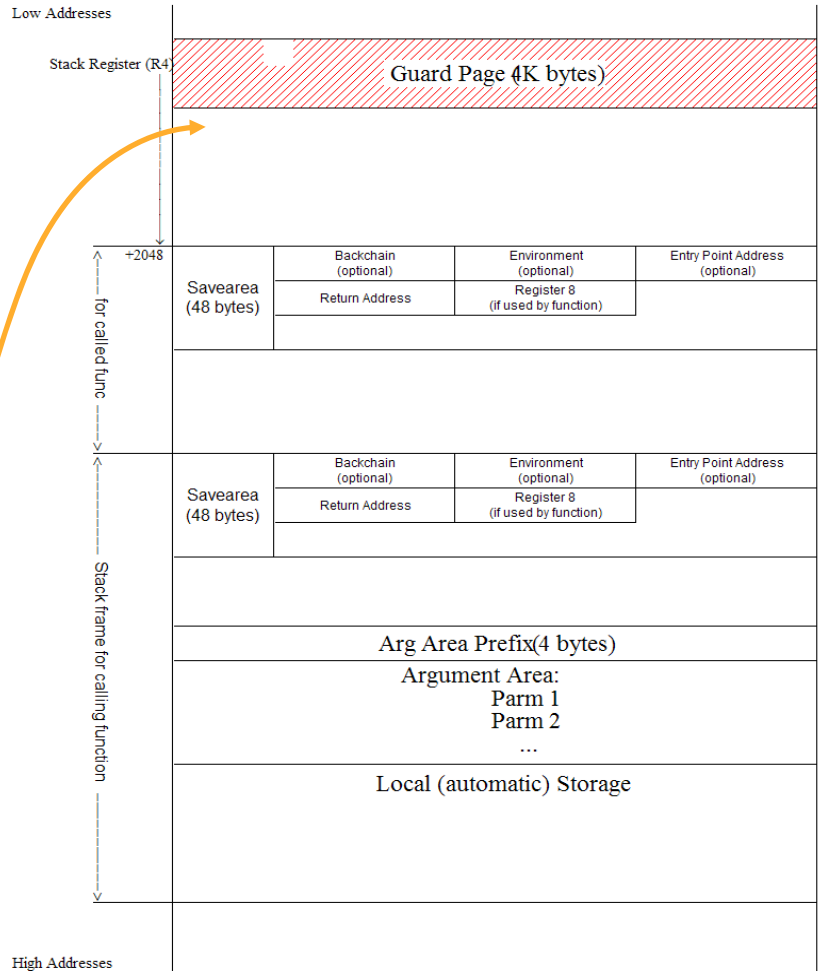

```
void f1(void) {
   f2();
   nop  flag
};

void f2(void) {
   // ...
   //
   // ...
};
```

```
000048  9067  4788        STM   r6,r7,1928(r4)
00004C  A74A  FF80        AHI   r4,H'-128'
```

1. Construct arguments in current stack frame's argument area
2. Pick up addresses of function's "environment" and code from the function descriptor
3. Link (BASR or BRAS) to called function
4. **Store registers used by called function**

   if the previous stack frame is close to the guard page this store (STM) may touch the guard page, causing an *interrupt that will be handled by the run time;* who, in turn, will extend the stack

XPLink Function Call Example




```
void f1(void) {
   f2();
   nop  flag
};

void f2(void) {
   // ...
   //
   // ...
};
```

```
000048  9067  4788        STM   r6,r7,1928(r4)
00004C  A74A  FF80        AHI   r4,H'-128'
```

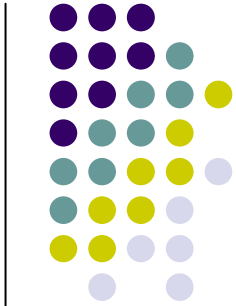
1. Construct arguments in current stack frame's argument area
2. Pick up addresses of function's "environment" and code from the function descriptor
3. Link (BASR or BRAS) to called function
4. Store registers used by called function
5. **Update Stack Register**

   the stack register may (or may not) now point into the guard page; this is of no concern for this particular call

Ideal XPLink Applications

- Highly-modular with many calls to small functions
- Minimal number of calls between XPLINK and NOXPLINK-compiled functions (which requires expensive stack switching glue code) -- **not applicable to 64-bit**
  - XPLink cross-linkage support is provided in C/C++ Compiler and for High Level Assembler (via macros)
  - In general, you cannot bind XPLINK-compiled and NOXPLINK-compiled functions together in the same program object -- **not applicable to 64-bit**
- XPLink requires Binder, and the output executable must reside in a PDSE or the UNIX file system.

Non-Ideal XPLink Applications

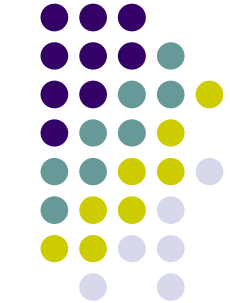


- The following can degrade performance or otherwise make using XPLink unattractive:
  - **31-bit only issues**
    - Large number of cross-linkage calls between XPLink and non-XPLink functions (requires stack switching glue code)
      - In an XPLink environment the C RTL is compiled XPLINK so non-XPLink callers of C functions go through stack switching glue
      - The C RTL uses stack switching glue code internally
        - Hex Math Library requires stack switch to run on Upstack (IEEE Floating Pt Library is ok, it's XPLink)
  - **31-bit and 64-bit issue**
    - Using unsupported environment or function (more on this later)

Cross-linkage Function Calls

- Since XPLINK and NOXPLINK-compiled parts cannot be mixed in the same program object, the DLL calling mechanism is the primary method for calling between XPLink and non-XPLink
  - Also supported are fetch()/fetchep() and LE's CEEFETCH macro
- The following do *not* support calls to XPLINK-compiled functions:
  - COBOL Dynamic Call
  - PL/I Fetch
  - CEELOAD (i.e.. the traditional LOAD/BALR)

Cross-linkage Function Calls



Ok, so there is *some* support for calling non-XPLink functions statically from XPLink:

- **#pragma linkage(..., OS_NOSTACK)**
  - **In 31-bit**, this the identical to OS31_NOSTACK
  - Generates direct call using OS Linkage conventions (no glue, so fast) but only 72-byte (31-bit)/144-byte (64-bit) savearea (e.g. C headers)
- **#pragma linkage(..., OS_UPSTACK) -- 31-bit only**
  - Generates call to RunOnUpStack glue so called function gets control with OS Linkage conventions and LE-conforming stack
- The intent is to be able to call Assembler "leaf" routines to perform functions not easily done from C/C++.

Stack Switching Glue Code

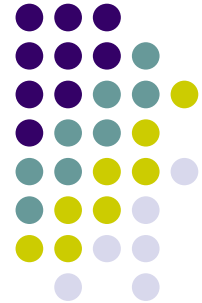
- Calls between XPLink and non-XPLink functions require LE to insert "glue code" that will:
  - switch between the upward and downward growing stacks
  - adjust parameter list formats
    - non-XPLink uses R1 pointing to list of parameters (or parm addresses)
    - XPLink passes parameters in general and floating point registers

Stack Switching Glue Code

- Services provided to allow non-XPLink and XPLink routines to run on the correct upward- or downward-growing stack:
  - **CEEVROND (RunOnDownstack)** -- calls XPLINK-compiled function from non-XPLink caller
  - **CEEVRONU (RunOnUpstack)** -- calls NOXPLINK-compiled function from XPLink caller
  - **CEEVH2OS (XPLink-to-OSLinkage)** -- calls non-XPLink function from XPLink callers using OS Linkage conventions

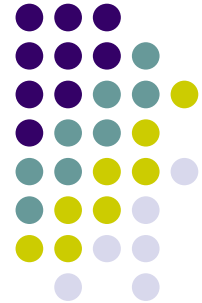
Refer to the z/OS Language Environment Vendor Interfaces book for details on these CWIs (Compiler-Writer Interfaces).

Unsupported Environments



- ~~CICS~~® (added in TS 3.1)
- ~~DB2~~® ~~stored procedures~~ (EXEC SQL is allowed)
- ~~IMS~~™ ~~transactions~~ (calls to ctdli() allowed)
- ~~PIPI~~ (added in z/OS v1r3)
- PICI
- LRR
- AMODE-24 and non-LE conforming applications
- Child nested enclave must match parent enclave's XPLINK run-time option
- CEEBXITA and CEEBINT User Exits cannot be coded as XPLINK functions
- 64-bit additional limitations
  - AMODE-31 and non-LE conforming applications
  - Nested enclaves

Building an XPLink Application

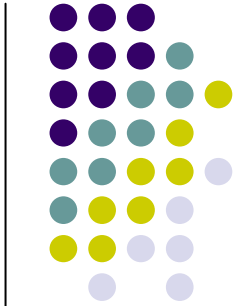


	Non-XPLINK...	XPLINK...
SYSLIB static libraries	SCEELKED SCEELKEX SCEEOBJ SCEECPP	SCEEBND2 ~~SCEEBIND~~
Dynamic Link Library (DLL) side decks	None for LE	In SCEELIB: 31-bit: ► CELHS003 (C RTL) ► CELHS001 (LE AWIs) ► CELHSCPP (C++) 64-bit: ► CELQS003 (C/LE RTL) ► CELQSCPP (C++)

Building an XPLink Application

- SCEEBND2 is a new LE data set containing XPLINK-compiled static routines ("stubs")
  - There are only a few
  - This data set can only be used with XPLINK applications (SCEELKED, etc. are non-XPLINK only)
- SCEELIB is a new LE data set containing LE DLL side decks
  - For XPLINK applications, the C RTL *is* a DLL

Building an XPLink Application

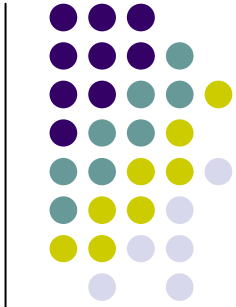


XPLINK Compile Option

NOXPLINK | XPLINK (optional suboptions)

- XPLINK(BACKCHAIN | NOBACKCHAIN)
  - With BACKCHAIN, STM instruction in prolog begins with register 4 to provide an explicit link between stack frames. This is not necessary for tools like CEEDUMP and slows down the prolog code.
- XPLINK(STOREARGS | NOSTOREARGS)
  - With STOREARGS, compiler inserts extra code after prolog to explicitly store parameter registers into argument area in caller's DSA.

Building an XPLink Application

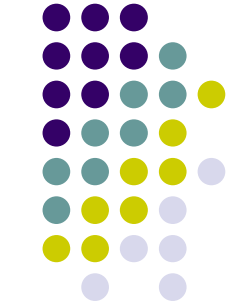


XPLINK Compile Option

NOXPLINK | XPLINK (optional suboptions)

- XPLINK(OSCALL(Downstack | Upstack | Nostack))
  - Alters default behavior of #pragma linkage(..., OS)
- XPLINK(NOGUARD | GUARD)
  - NOGUARD will generate an explicit check of the stack floor in the prolog code
- XPLINK(NOCALLBACK | CALLBACK) **z/OS R2**
  - CALLBACK will allow non-XPLink function pointers or descriptors to be correctly used in an XPLink program. __callback qualifier is preferred.

Building an XPLink Application



c89 changes

- The XPLINK compile option can be specified as:
  - -Wc,xplink (-Wc,goff is forced)
  - -Wc,lp64 (-Wc,xplink,arch(5+) forced, -Wc,float(ieee) defaulted)
  - Object files are still Fixed 80, but now in GOFF format
- A new XPLINK linkedit option is also required (this option is *not* passed to the binder):
  - -Wl,xplink
  - -Wl,lp64
  - Tells c89 to use SCEEBND2 and SCEELIB data sets
  - Forces binder options DYNAM=DLL and CASE=MIXED (required for calls to C RTL and other DLLs)

Building an XPLink Application

c89 simple example

- XPLink "Hello World" example:

```
c89 -o HelloWorld -Wc,xplink -Wl,xplink HelloWorld.c
```

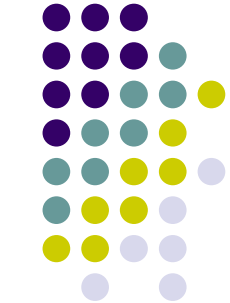
- 64-bit "Hello World" example:

```
c89 -o HelloWorld -Wc,lp64   -Wl,lp64   HelloWorld.c
```

Running an XPLink Application

- **XPLink Requires that both LE run-time libraries are available at execution time:**
  - SCEERUN
  - SCEERUN2
    - It's a PDSE (required by XPLink)
    - Contains XPLink versions of C RTL, locales and converters, etc.
      - ☞31-bit: CELHV003, CELHDCPP (C++)
      - ☞64-bit: CELQLIB (combined C/LE RTL), CELQDCPP (C++)

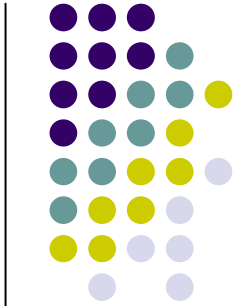
Running an XPLink Application



New LE Run-Time Option - 31-bit

- XPLINK(ON|OFF)
  - XPLINK(OFF) is the default
  - If main() is compiled XPLINK, then the XPLINK run-time option is forced ON
  - If main() is compiled NOXPLINK *but* calls an XPLINK-compiled function, then XPLINK(ON) must be specifed, otherwise error message CEE3555S will be generated and the application is terminated
  - Cannot be specified in CEEDOPT as a system installation default, the XPLINK run-time option must be specified on an application by application basis (when needed)

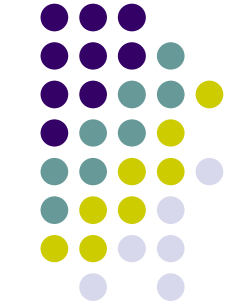
Running an XPLink Application



Changed LE Run-Time Options – 31-bit

- STACK (usinit_size, usinc_size, ANY|BELOW, KEEP|FREE, dsinit_size, dsinc_size)
  - STACK suboptions:
    - upstack initial size, upstack increment size
    - upstack location (ANY | BELOW)
      - **location ANY forced when XPLINK(ON) in effect**
    - duration (KEEP | FREE)
    - **downstack initial size <-- new**
    - **downstack increment size <-- new**
  - Downstack sizes do not include storage for guard page
  - Downstack not allocated in XPLINK(OFF) environment

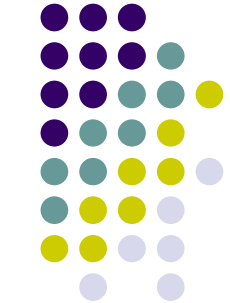
Running an XPLink Application



Changed LE Run-Time Options – 31-bit

- THREADSTACK (usinit_size, usinc_size, ANY|BELOW, KEEP|FREE, dsinit_size, dsinc_size)
  - THREADSTACK suboptions:
    - –upstack initial size, upstack increment size
    - –upstack location (ANY | BELOW)
      - **location ANY forced when XPLINK(ON) in effect**
    - –duration (KEEP | FREE)
    - **–downstack initial size <-- new**
    - **–downstack increment size <-- new**

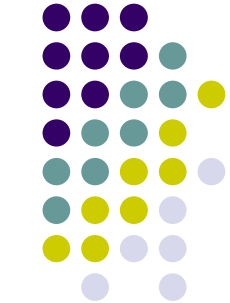
Running an XPLink Application



Changed LE Run-Time Options – 31-bit

- THREADSTACK continued...
  - Downstack sizes do not include storage for guard page
  - Downstack not allocated in XPLINK(OFF) environment
  - ☞THREADSTACK option replaces the NONIPTSTACK and NONONIPTSTACK options (which are still accepted for compatibility)

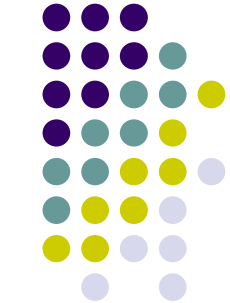
Running an XPLink Application



Changed LE Run-Time Options – 31-bit

- ALL31
  - When the XPLINK(ON) run-time option is in effect, the ALL31 run-time option will be forced to ON.
  - No AMODE 24 routines allowed in an XPLINK(ON) environment
- RPTSTG
  - Will report storage statistics for downward stack

Running an XPLink Application



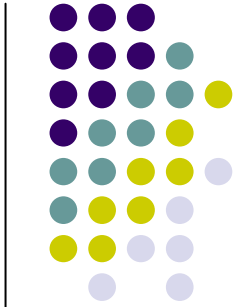
New LE Run-Time Options - 64-bit

- Many unneeded options eliminated (XPLINK, ALL31)
- Many old options have **64-bit** counterparts: ...
  - **HEAPPOOLS64**(OFF|ON,up to *12 cellsize,cellcounts)*

  [OFF,8,4000,32,2000,128,700,256,350,1024,100,
  2048,50,3072,50,4096,50,8192,25,16384,10,32768,5,65536,5]
    - ☞ In 31-bit, HEAPPOOLS takes cell percentages rather than cell counts.
  - **HEAP64**(init,incr,KEEP|FREE,
    init31,incr31, KEEP|FREE,
    init24,incr24,KEEP|FREE)

  [1M,1M,KEEP,32K,32K,KEEP,4K,4K,FREE]
    - ☞ __malloc31() and __malloc24() available for storage Below-The-Bar and Below-The-Line.

Running an XPLink Application



New LE Run-Time Options - 64-bit

- Many old options have **64-bit** counterparts:
  - **STACK64**(init,incr,max)

    [1M,1M,128M]
  - **THREADSTACK64**(OFF|ON,init,incr,max)

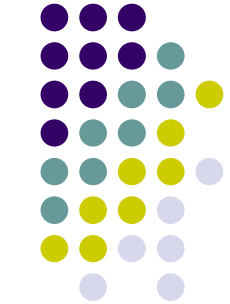
    [OFF,1M,1M,128M]

    ☞ Stack is *always contiguous* (not so in 31-bit).

    ☞ Current stack size identified by guard page.

    ☞ On stack overflow (exception) guard page is moved down.

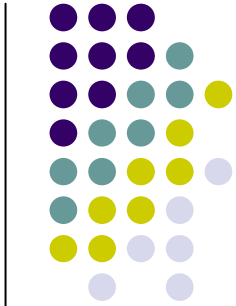
Compiler-writer Interfaces



The following CWIs are new for XPLink:

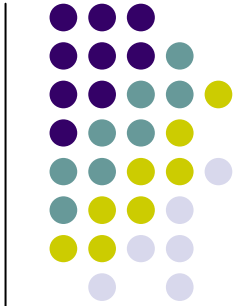
- Documented in LE Vendor Interfaces
- Declared in <edcwccwi.h> (in SCEESAMP data set)
  - __dsa_prev()
    - Takes as input the address and format of "current" DSA
    - Returns address of previous (logical or physical) DSA and its format
    - Call it in a loop to unwind the stack

Compiler-writer Interfaces



- __ep_find()
  - Takes as input a DSA address and format
  - Returns the address of the entry point of the function owning the input DSA
- __bldxfd() - **31-bit only**
  - XPLink environment only
  - Takes a function pointer (entry point) of unknown linkage as input
  - Returns a "function pointer" that can be called by all linkage types

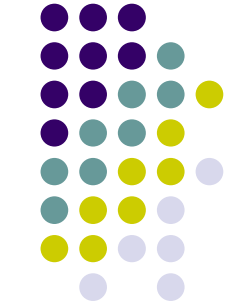
Appendix A
Sample Generated XPLink Code



There are also new XPLink-style entry points and Program Prolog Areas (PPAs)

```
                          00001 |       *  #include <stdio.h>
                          00002 |       *
                          00003 |       *  main() {

000020                                     @1L0     DS    0D               XPLink entrypoint marker
000020  00C300C5                                    =F'12779717'             '.C.E.E.1' eyecatcher
000024  00C500F1                                    =F'12910833'             (x'F1' == entry pt marker)
000028  FFFFFFE0                                    =F'-32'                  Offset to XPLINK-style PPA1
00002C  00000080                                    =F'128'                  DSA size
000000                    00003 |                   main     DS    0D               Function entry point
000000  9057  4784        00003 |                            STM   r5,r7,1924(r4)
000004  A74A  FF80        00003 |                            AHI   r4,H'-128'
000008                    End of Prolog

                          00004 |       *      printf("Hello world\n");
000008  5810  4804        00004 |                            L     r1,#Save_ADA_Ptr_1(,r4,2052)
00000C  9856  1010        00004 |                            LM    r5,r6,=A(printf)(r1,16)
000010  0D76              00004 |                            BASR  r7,r6
000012  4700  0003        00004 |                            NOP   3
                          00005 |       *  }
000016  4130  0000        00005 |                            LA    r3,0
00001A                    00005 |                   @1L1     DS    0H

00001A                    Start of Epilog
00001A  5870  480C        00005 |                            L     r7,2060(,r4)
00001E  4140  4080        00005 |                            LA    r4,128(,r4)
000022  07F7              00005 |                            BR    r7
```

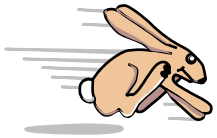
Appendix B
Main XPLink and 64-bit Publications of Interest



- LE Programming Guide (SA22-7561) has a chapter on developing XPLink applications.
- LE Programming Guide for 64-bit (SA22-7569)
- LE Vendor Interfaces (SA22-7568) has new detailed description, new CWIs, and "all" 3 LE-conforming linkages:
  - Standard LE linkage (includes COBOL, PL/I, etc)
  - C++ Fastlink
  - XPLINK
- LE Writing Interlanguage Communication Applications (SA22-7563)
- LE Debugging Guide (GA22-7560)
- XPLINK Redbooks™ (http://publib-b.boulder.ibm.com/abstracts/sg245991.html)
- 64-bit Redpaper (http://publib-b.boulder.ibm.com/abstracts/redp9110.html)
- C/C++ for z/OS books updated too

Appendix C

Performance Measurements - 31-bit



Reference Materials

XPLink Performance Redbook

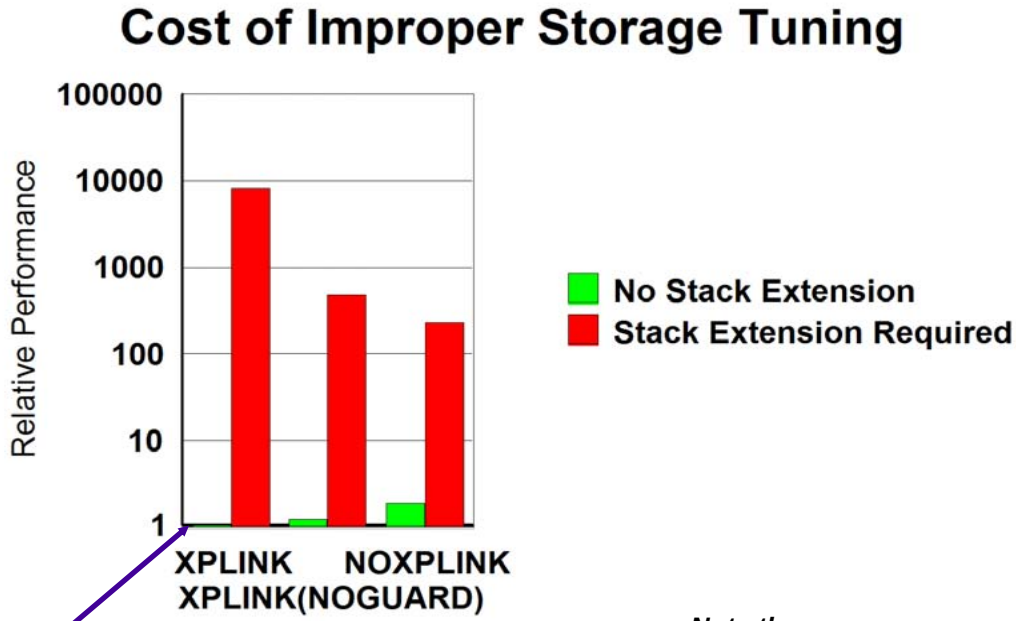
- Measurements made over summer 2000
- SG24-5991

  www.redbooks.ibm.com/redbooks/SG245991.html

- Measurements were made on shared systems in Toronto (a development system) and Poughkeepsie (the ITSO system)
- Results were generally repeatable within 1-2%
- Highlights are reported here, details are in the Redbook

The Importance of Storage Tuning

- Stack overflow detection is by program check
  - Improperly-tuned stack allocation can cause disastrous performance
  - use XPLINK(NOGUARD) in portions of the application where stack growth is unpredictable

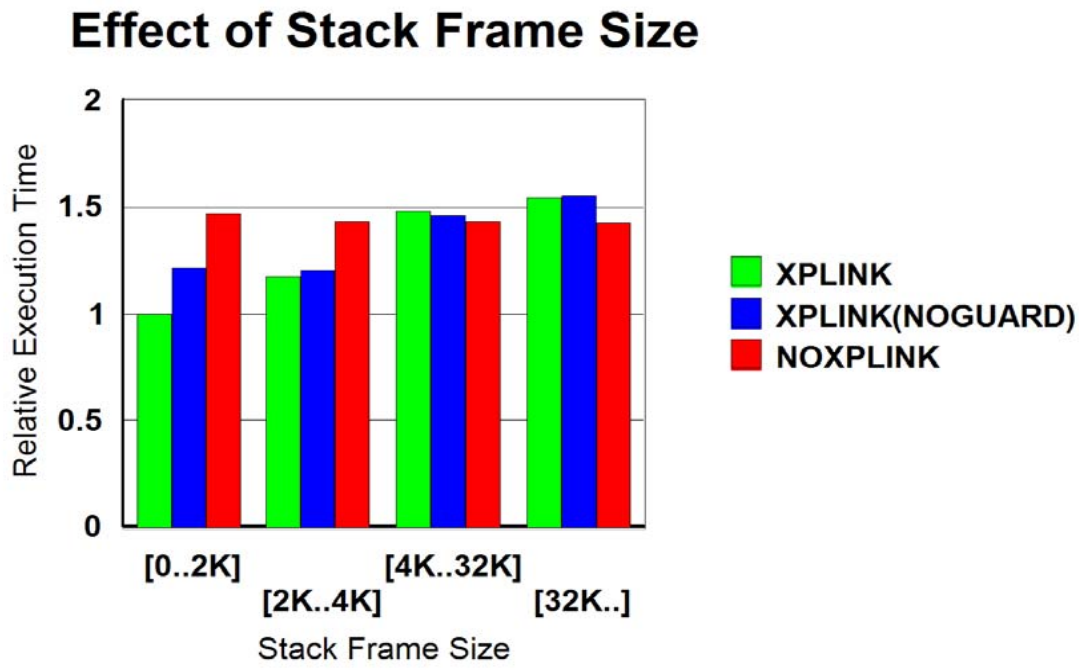


Baseline (1) is the XPLINK test running with a "proper" initial stack allocation

Note the logarithmic scale

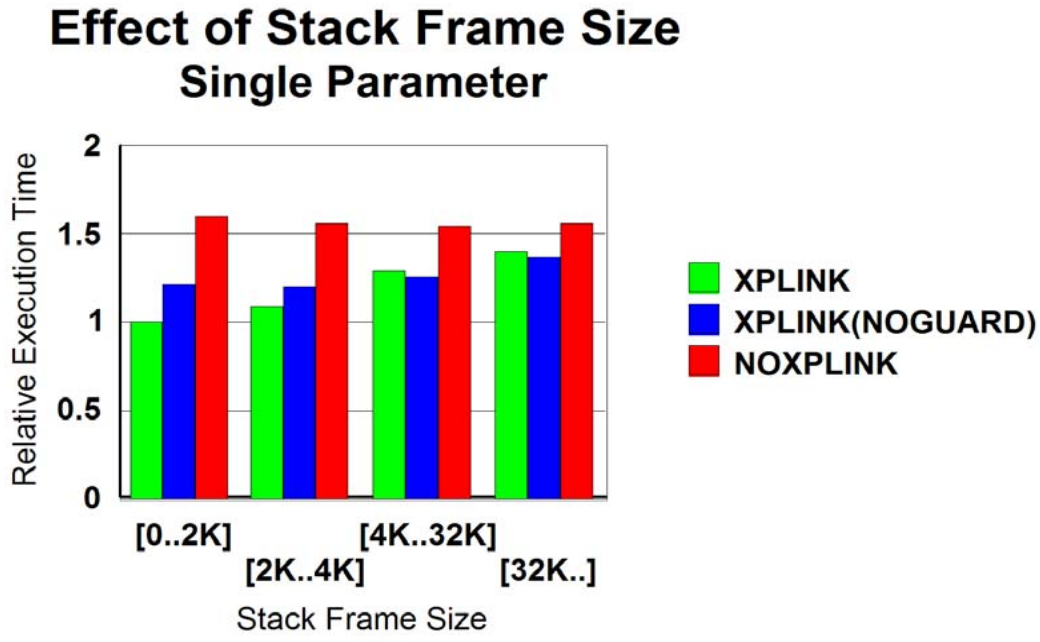
Effect of Stack Frame Size

- XPLink is optimized for small stack frames (that is, small amounts of automatic storage)

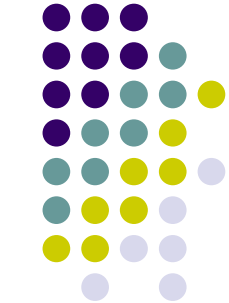


Effect of Number of Parameters

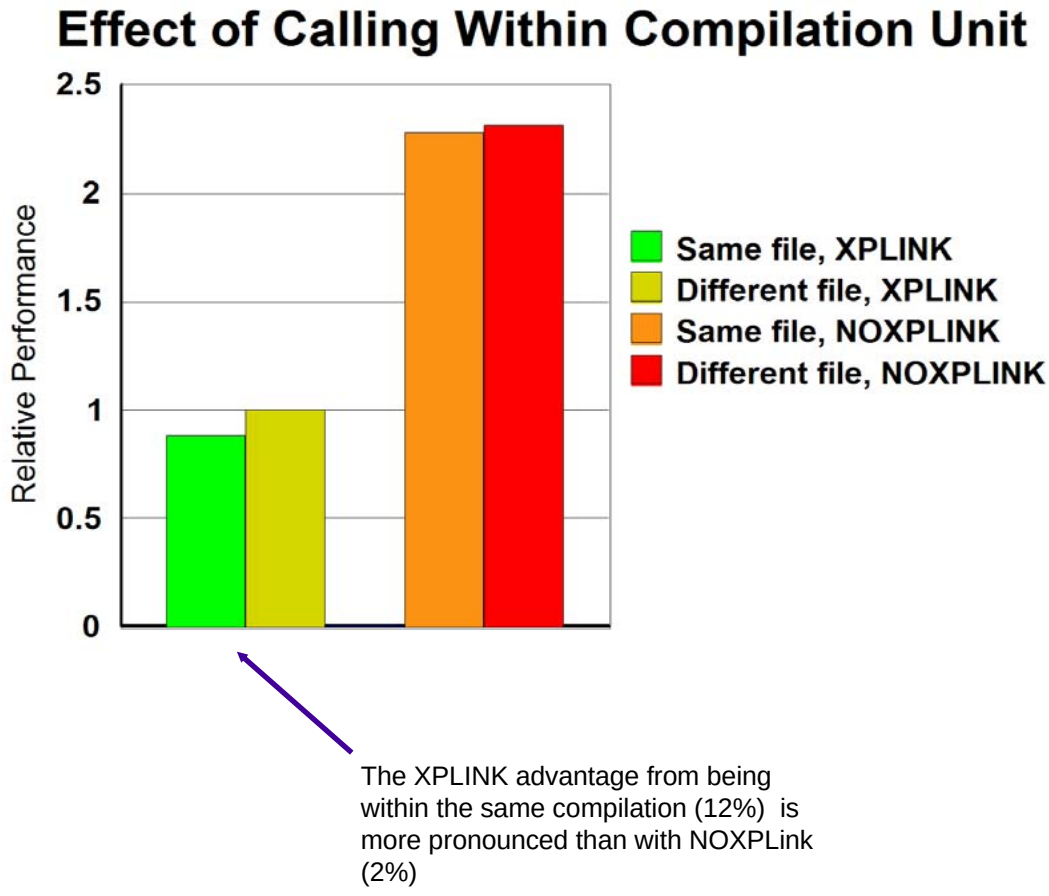
- XPLinkfunction prologs change with the number of function arguments
  - Fewer arguments gives better code-generation opportunities
  - The worst case (>32K local storage, 1 parameter) is better than non-XPLink



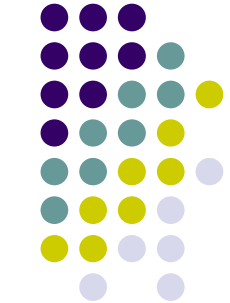
Calling Within a Compilation Unit



- Calls are often faster when made to a function in the same compilation unit
  - Environment pointer (WSA pointer for NOXPLINK) is often the same
  - Can be called with relative branch instructions in XPLink
    - function is entered with no base register

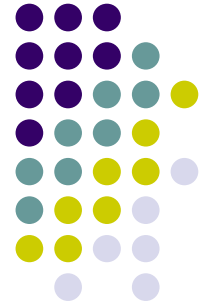


Mixing XPLink with a COBOL Application

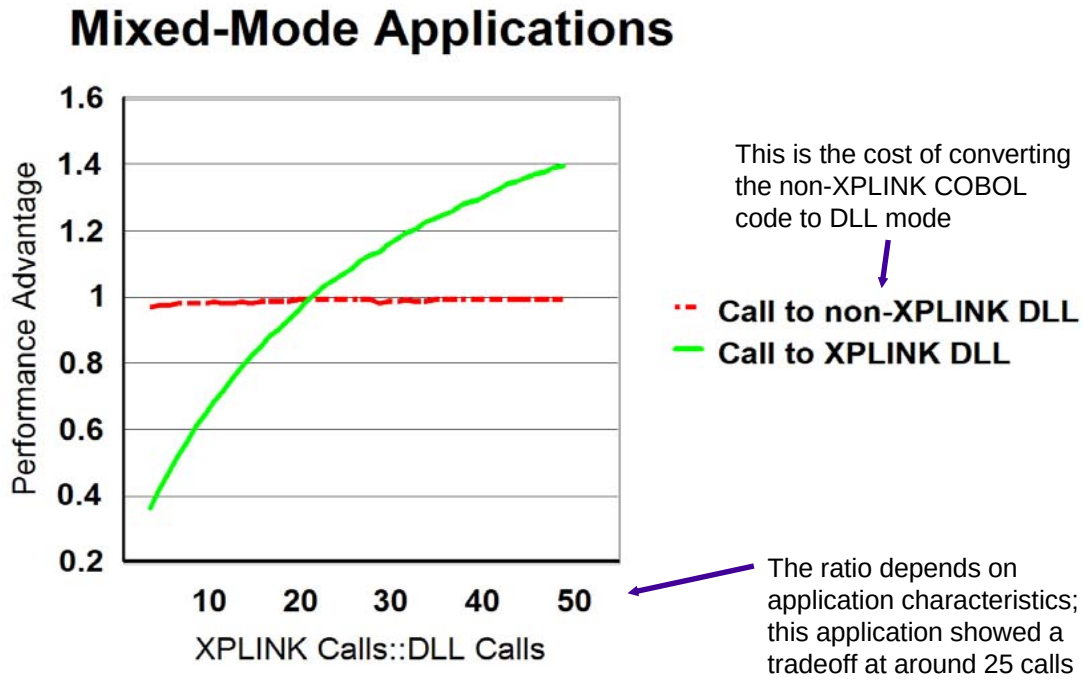


- COBOL does not support XPLink
- Separate the XPLink (C/C++) code from the non-XPLink code
- Put XPLink code into a DLL

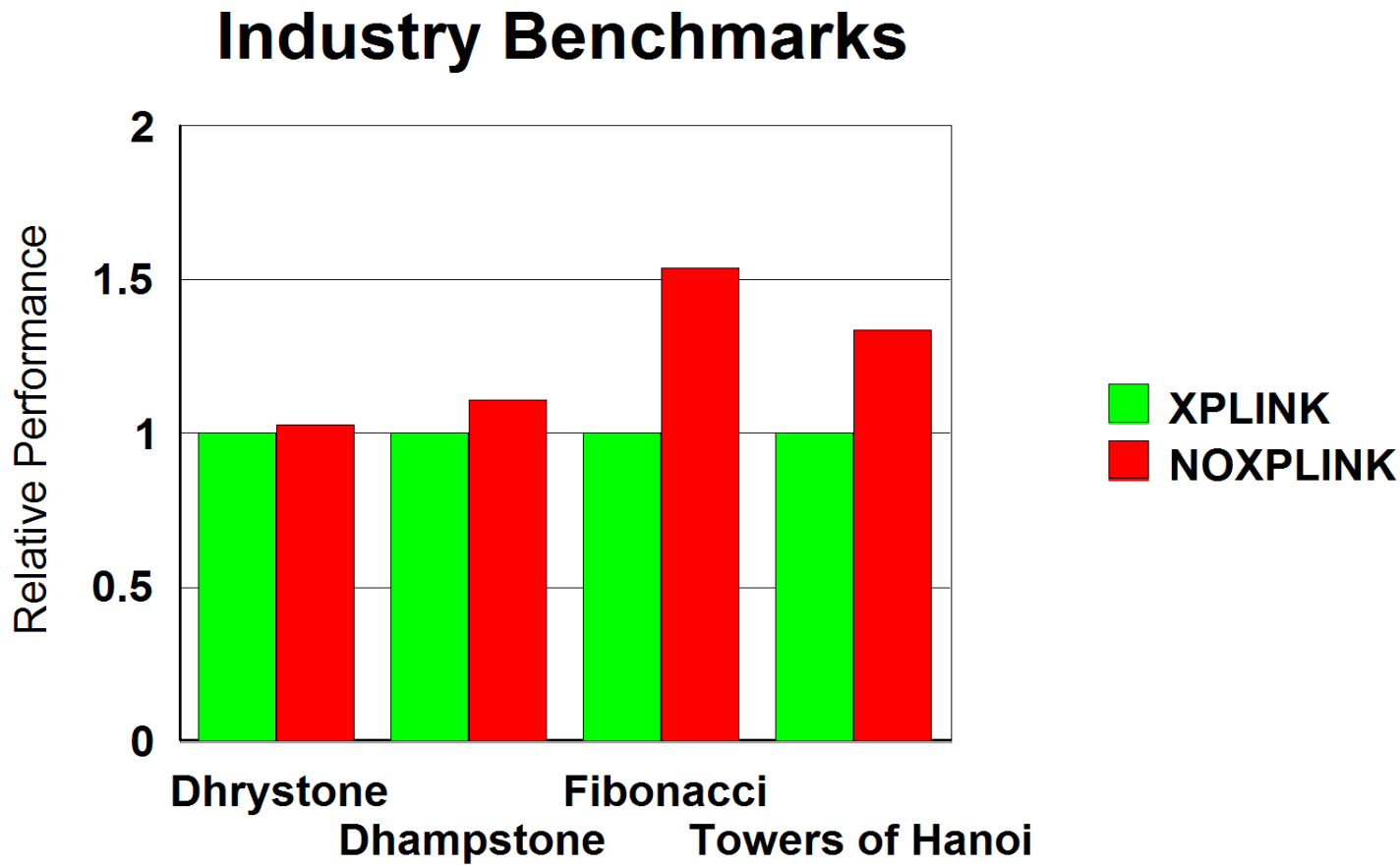
Mixing XPLink with a COBOL Application



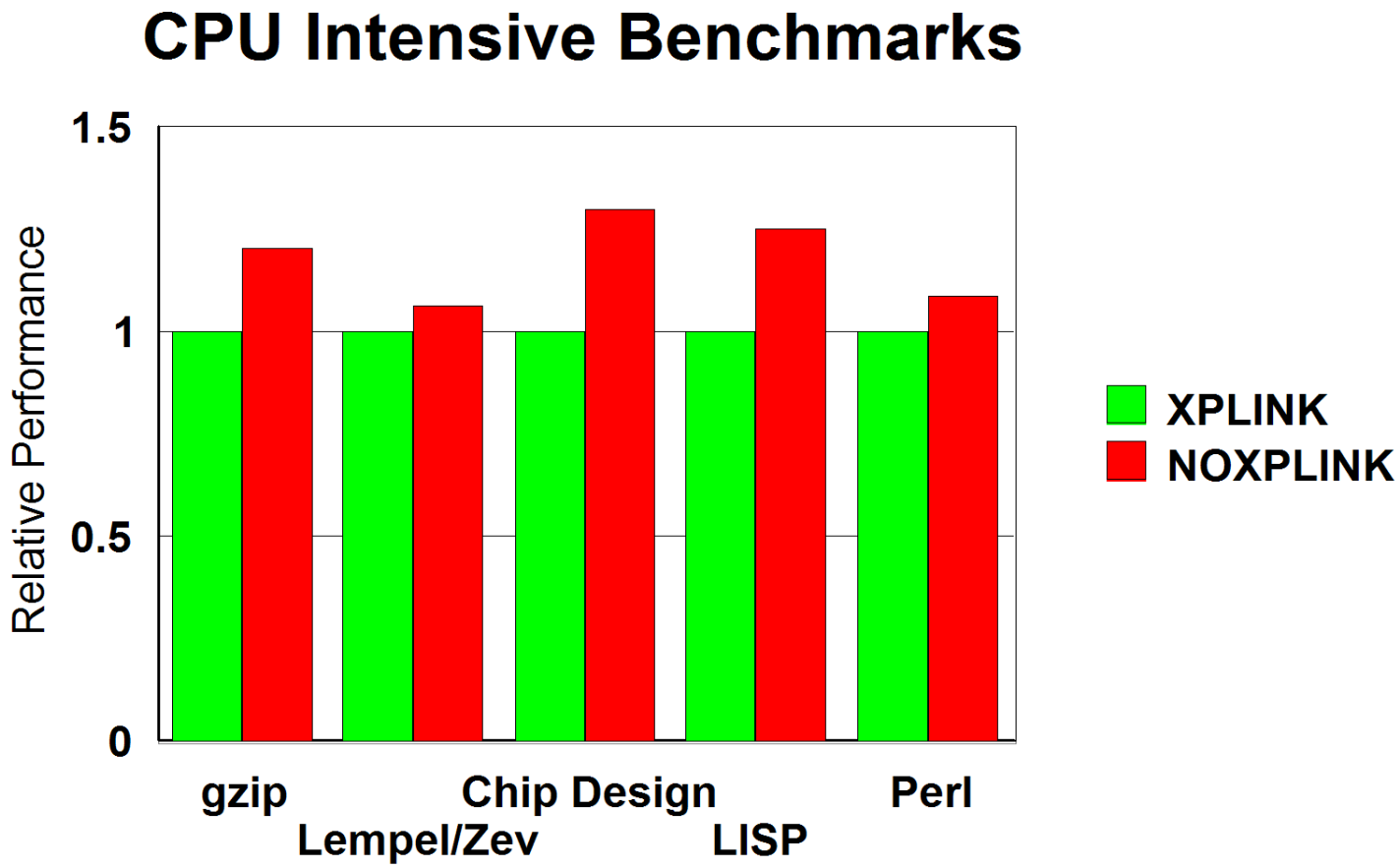
- Overall application performance depends on the number of calls ***inside*** the DLL for every call ***into*** the DLL
- This is typical of the performance characteristics expected from a C/C++ DLL written for use with a COBOL DLL.



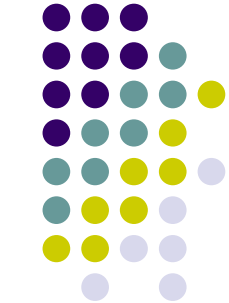
Industry Benchmarks



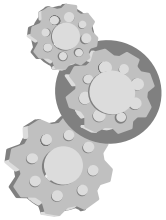
CPU Intensive Benchmarks



Appendix D

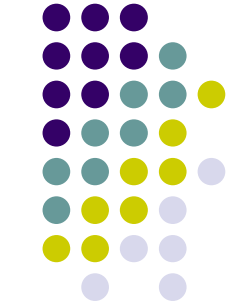


XPLink More Details



Reference Materials

Prolog Comparison for Large Automatic Storage



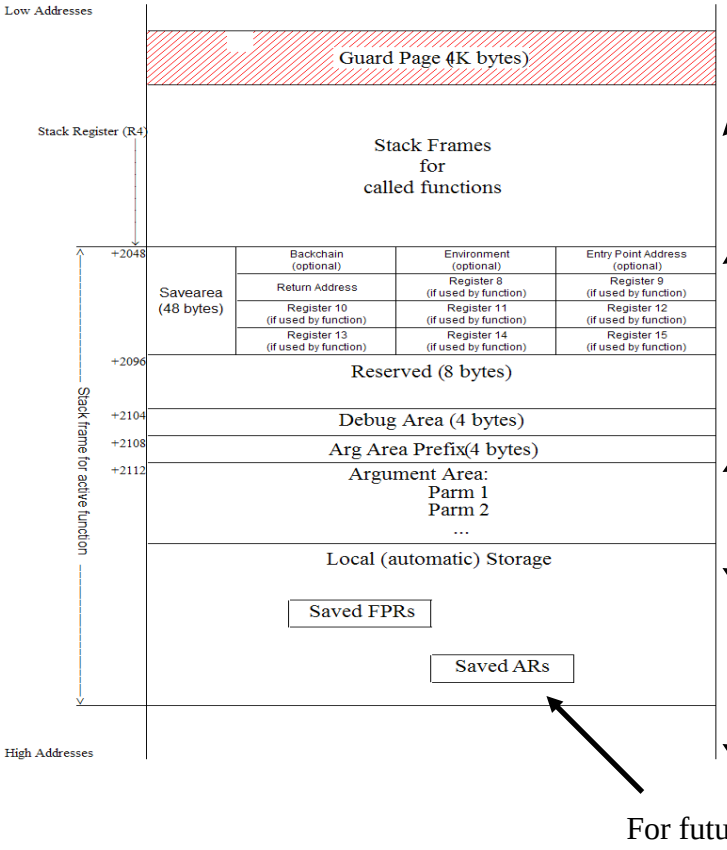
- Functions with large automatic storage clearly do not get the same performance advantage with XPLink

```
000060  47F0  F022        B     34(,r15)
000082  90E4  D00C        STM   r14,r4,12(r13)
000086  58E0  D04C        L     r14,76(,r13)
00008A  5800  F008        L     r0,8(,r15)
00008E  1E0E              ALR   r0,r14
000090  5500  C314        CL    r0,788(,r12)
000094  4130  F03C        LA    r3,60(,r15)
000098  4720  F014        BH    stack extender
00009C  58F0  C280        L     r15,640(,r12)
0000A0  90F0  E048        STM   r15,r0,72(r14)
0000A4  9210  E000        MVI   0(r14),16
0000A8  50D0  E004        ST    r13,4(,r14)
0000AC  18DE              LR    r13,r14
```

```
void f1(void) {
   f2(1,2,3);
   NOP flag
};

void f2(int i,int j, int k)
{
   // huge local storage
   // requirements
   // ...
   //
   // ...
};
```

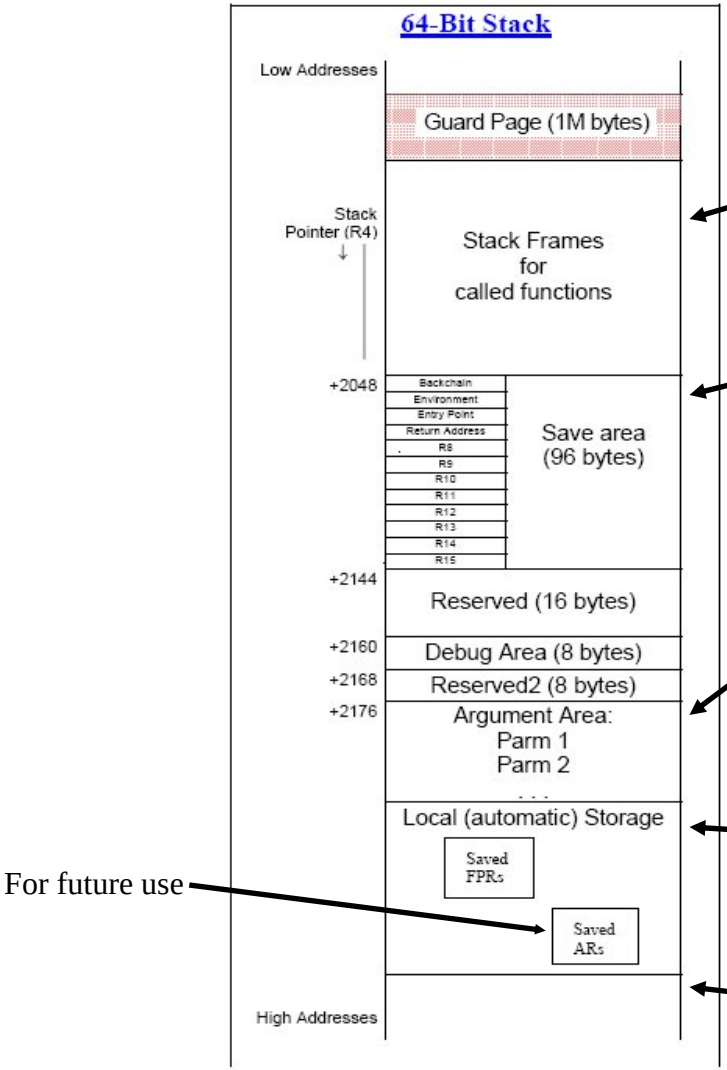
```
000048  9023  4844        STM   r2,r3,2116(r4)
00004C  1804              LR    r0,r4
00004E  0D20              BASR  r2,0
000050  A72A  0034        AHI   r2,H'52'
000054  5A40  2000        A     r4,0(,r2)
000058  5940  C364        C     r4,868(,r12)
00005C  A744  0022        JL    stack extender
000060  9058  4804        STM   r5,r8,2052(r4)
000064  5000  4800        ST    r0,2048(,r4)
000068  1882              LR    r8,r2
00006A  1820              LR    r2,r0
00006C  5820  2844        L     r2,2116(,r2)
```

Stack Layout Detail - 31-bit



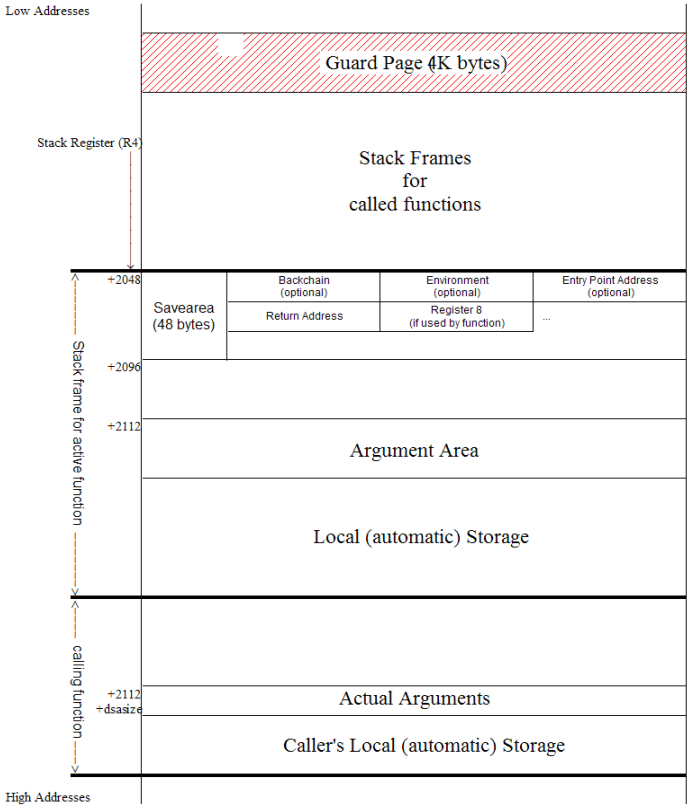
- Current Register 4
  - possibly in guard page, possibly not
- Registers on entry to active function (**48 bytes in 31-bit 96 in 64-bit**)
- Argument area for calls in active function, large enough to accommodate biggest actual argument list in function
- Automatic storage for active function
- Stack frame for caller

Stack Layout Detail - 64-bit



- Current Register 4
  - possibly in guard page, possibly not
- Registers on entry to active function
- Argument area for calls in active function, large enough to accommodate biggest actual argument list in function
- Automatic storage for active function
- Stack frame for caller

Argument Addressability



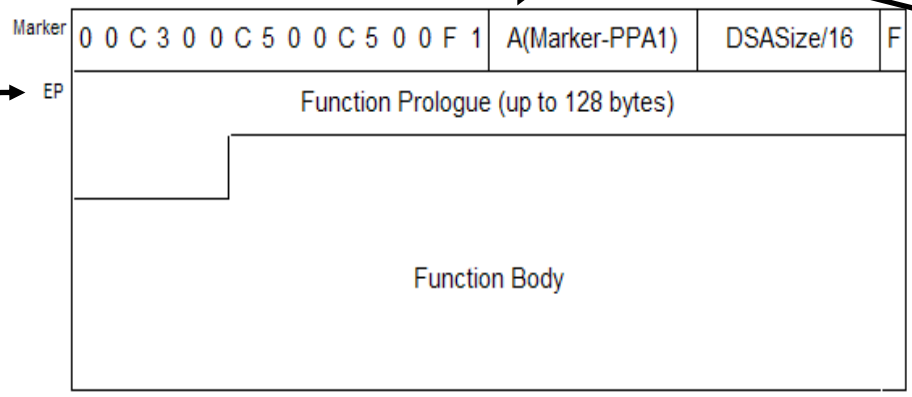
- Active Stack Frame
- Actual Arguments to Active function, built by caller in caller's argument area. Addressable at:

31-bit (R4+2112+*active stack frame size*)

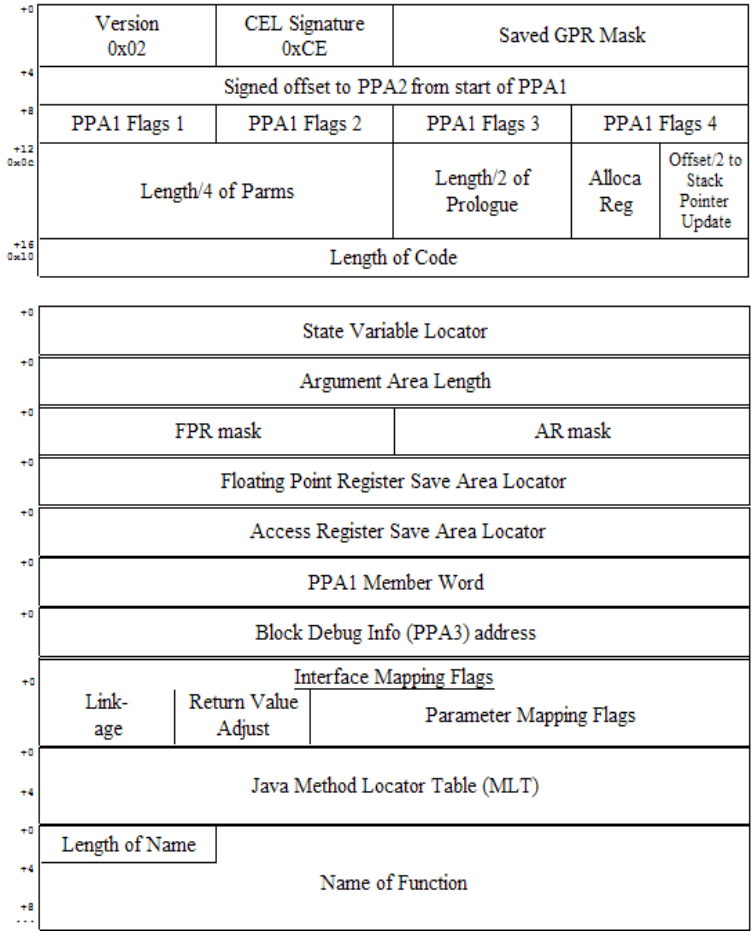
64-bit (R4+2176+*active stack frame size*)

Entry Point Marker

- Entry Point
  - doubleword aligned
- Entry Point Marker
  - 16 bytes before entry point, doubleword aligned
  - Shows up in dump as **`.C.E.E.1`**
- PPA1 Locator



New PPA1 format

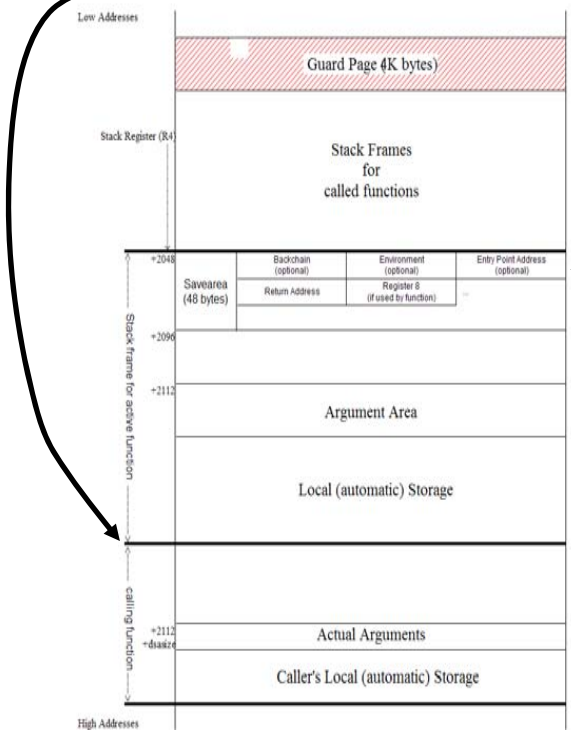


"Locators" are 4-bit register numbers followed by 28-bit offsets. Add the contents of the specified register and the specified offset to get to the target of the locator

- Fixed portion
- Optional Fields, presence indicated by flags in fixed portion

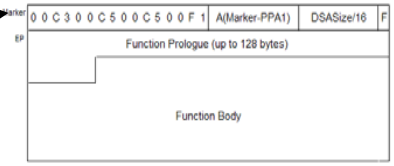
Stack Walking

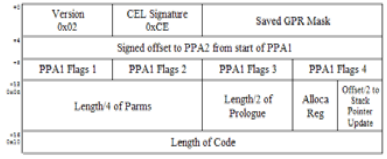
The Stack:

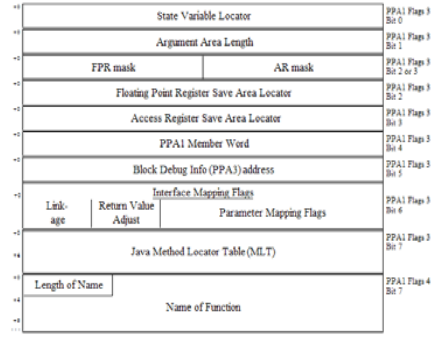



```
void f1(void) {
  f2();
     nop  flag
};

void f2(void) {
  // ...
  //
  // ...
};
```

The Code:





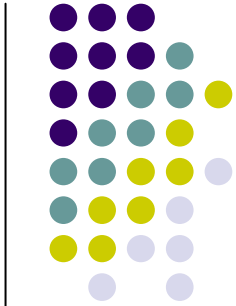


Stopped here, in dump, debugger, service routine &c

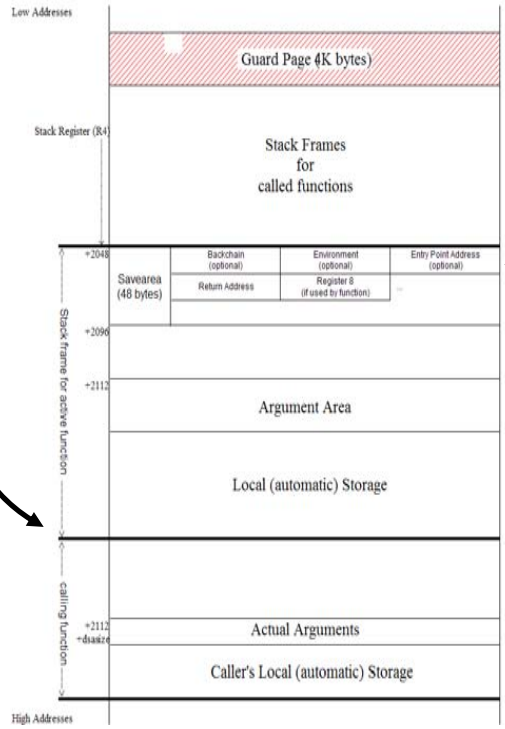
The steps described here assume we are ***not*** in f2()'s prolog. We can determine this by:

1. scanning backwards up to 128 bytes looking for the doubleword-aligned Entry Point Marker.
2. from the Entry Point Marker locating the PPA1 (as described in the following foils) and, in the PPA1, the length of the prologue and the offset of the instruction updating the stack pointer

Stack Walking Steps



The Stack:




```
void f1(void) {
  f2();
     nop  flag
};

void f2(void) {
  // ...
};
```

1 Pick up f2()'s return address from current stack frame

2 Look at call type

3 Pick up f2()'s entry point from current stack frame (computed in case of a relative call)

extract offset from the relative branch instruction just prior to the return point, add it to return address

4 Pick up f2()'s PPA1 offset

5 Locate f2()'s PPA1, examine f2()'s GPR Save Mask in the PPA1

this tells us which registers were saved in f2()'s stack frame by f2()'s prologue

similar rules apply to floating point registers: there's a "Floating Point Register Save Area Locator" in PPA1 which tells us where to find the FPR save area in the current stack frame, and the FPR mask which tells us which floating point registers were saved.

6 Store unsaved registers into f2()'s Save Area

the first time through (while processing the stack frame for the active function) use the values actually in the registers at the time of interrupt; subsequently, use the register values stored in the previous stack frame

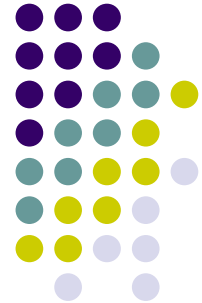
7 Pick up f2()'s dsasize and flags

8 Add f2()'s dsa size to current stack frame address to get f1()'s stack frame

if f2() uses alloca(), pick up the alloca register from the PPA1 and add the dsasize to that register

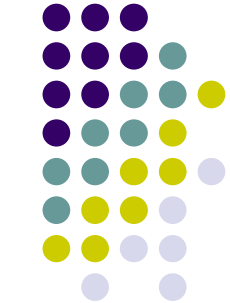
9 Repeat as required

Stack Walking Steps *(continued)*



- Stack structure is fully supported by
  - Debug Tool (**31-bit**)
  - dbx
  - IPCS LEDATA, described in z/OS MVS™ Interactive Problem Control System (IPCS) Commands
- Additional documentation can be found in
  - z/OS Language Environment Debugging Guide and Run-Time Messages
  - z/OS Language Environment Vendor Interfaces

Call Type Info and Call Descriptors

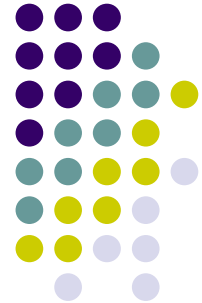


- Call Type Info: NOP(R) at call site
  - 31-bit uses NOP
  - **64-bit uses NOPR -- there is no call descriptor!**
- Used to describe return type and parameters passed in registers:

```
* f2();
      L      5,f2..env pointer
      BRAS   7,f2
      NOP    0(call type)        470t oooo
      ORG    *-2
      DC     H'(offset of EP marker or descriptor)/8'
      ...
* call descriptor (shareable)
      DS     0D
      DC     A(Signed offset to EP Marker)
      DC     AL1(return type)
      DC     AL3(parameter descriptor)
```

0000	Function is called with a BASR 7,6 instruction; register 6 will contain its entry point address
000l	Function is called via a BRAS 7,EP instruction; the called function does not have a base register on entry
0010	Reserved
0011	Reserved; the called function does not have a base register on entry
0100	Reserved
0101	Reserved
0110	Non-XPLink call inside XPLink function body
0111	Special linkage
1...	Reserved

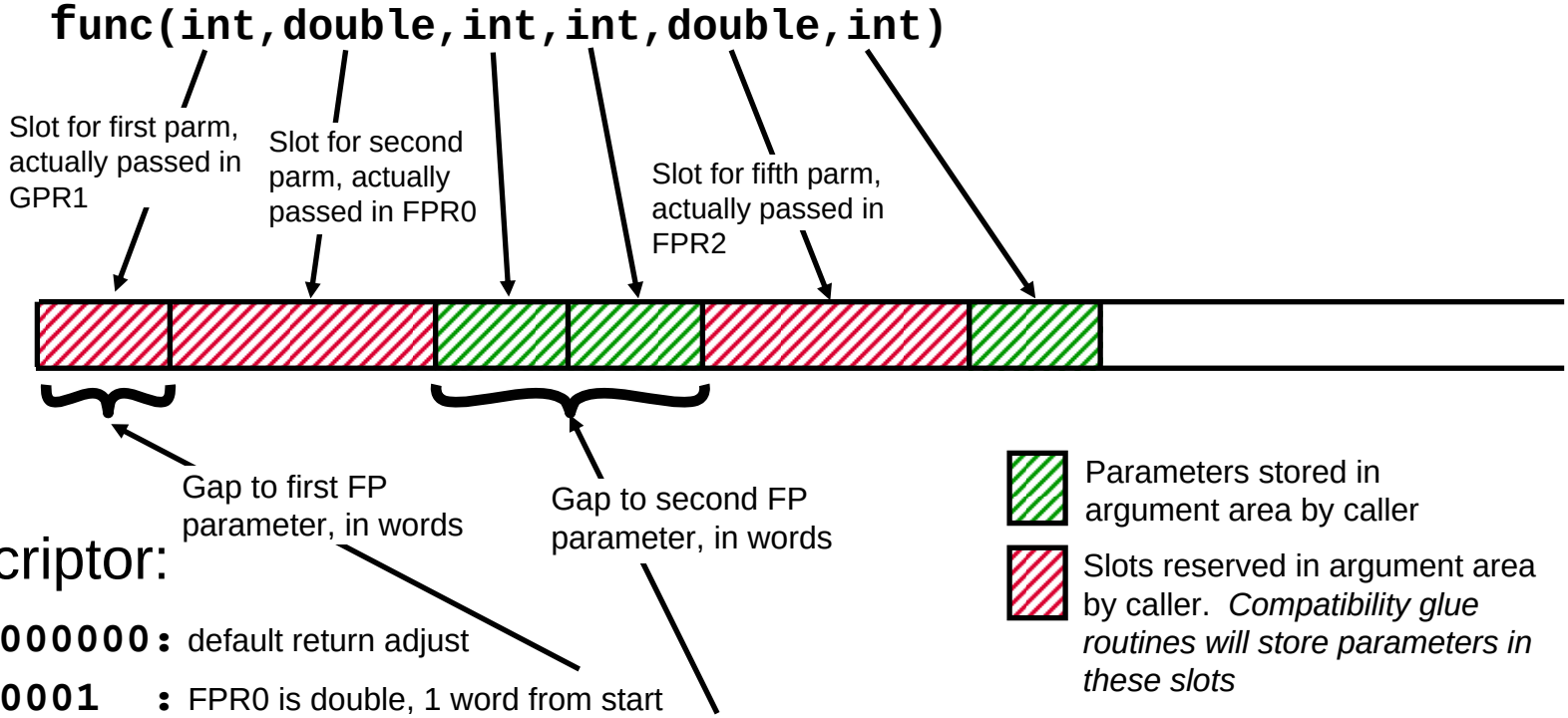
Call Descriptor - 31-bit



- Describes parameters locations for floating point parameters passed in registers
  - indicates where the "holes" in the stack-based parameter list occur
- Return adjust for return values in registers
- Solely for compatibility with non-XPLINK code

Floating Point Parameters, Example

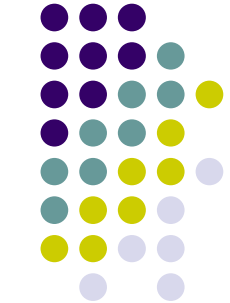
- Doubles passed in FPR0 and FPR2



Descriptor:

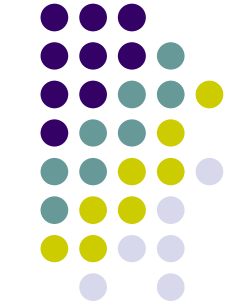
```
00000000: default return adjust
100001  : FPR0 is double, 1 word from start
100010  : FPR2 is double, 2 words from previous float
000000  : FPR4 not used
000000  : FPR6 not used
```

Examining Actual Arguments



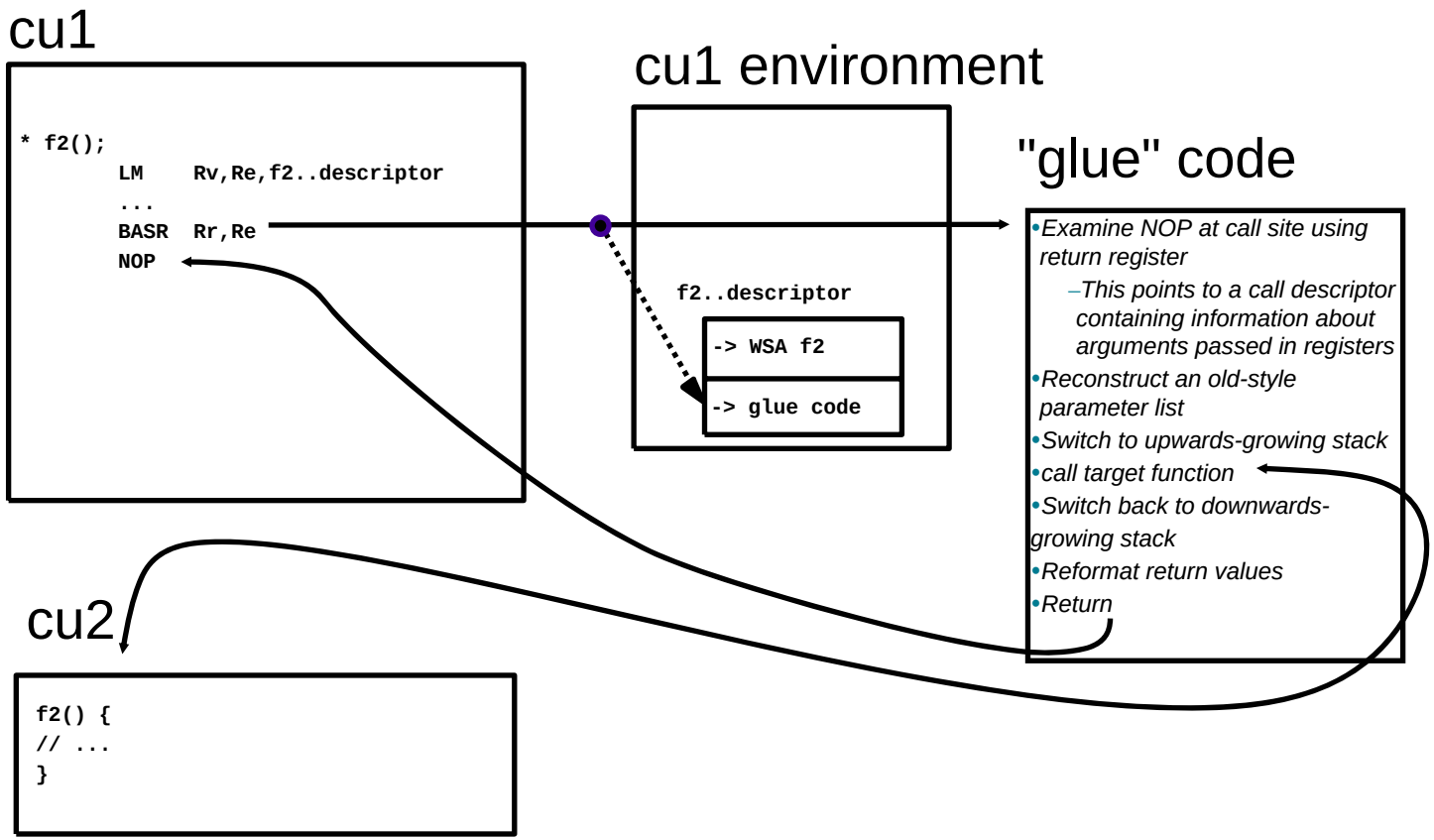
- Some arguments are passed in registers (GPRs 1-3, FPRs 0, 2, 4 ,6)
- Their values may have been lost if they are no longer required
- Compiler XPLINK(STOREARGS) option forces generated code to save incoming parameters in their natural places in the caller's argument area
- Even then, they may be lost if the called function modifies its arguments

New Linkage Specifier

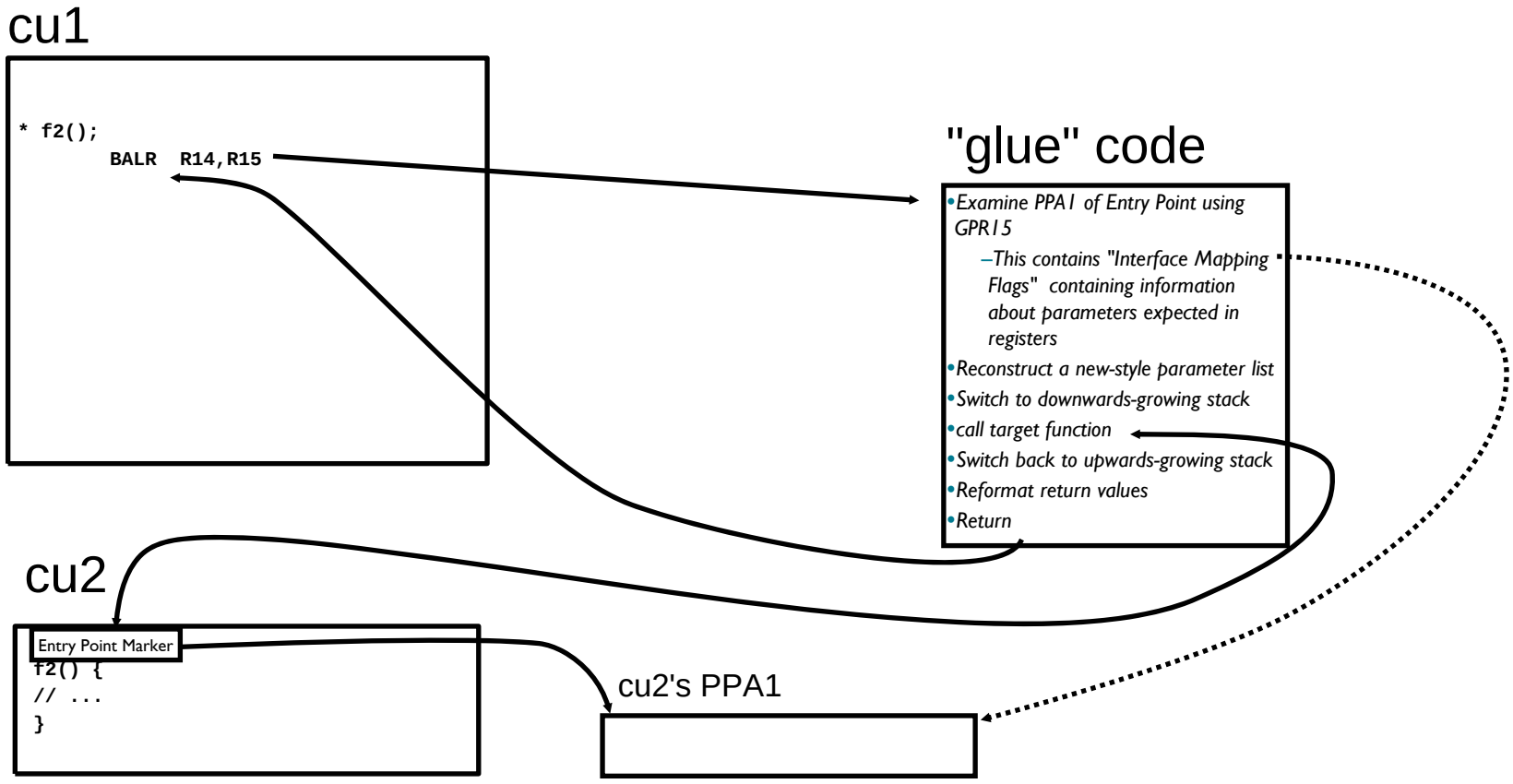


- New #pragma introduced to provide some low-level compatibility support
  - OS_NOSTACK, to call assembler code
    - R13 points to 18-word savearea
    - R14, R15 linkage
    - R1 points to parameters
    - No NAB, or other "Classic LE" stack artifacts
  - This is the default for linkage OS; it can be changed by a compiler option

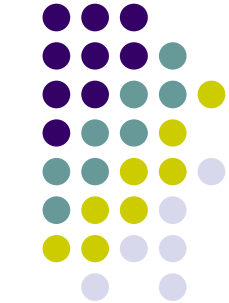
Compatibility: New Calling Old - 31-bit



Compatibility: Old Calling New - 31-bit



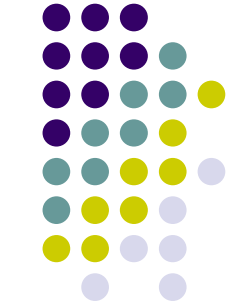
Appendix E



Debugging an XPLink Application

Reference Materials

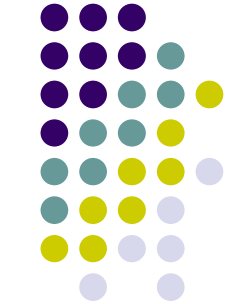
CEEDUMP Support for XPLink



- Traceback support for Up and Down stacks - **31-bit**

```
Traceback:
   DSA Addr   Program Unit   PU Addr      PU Offset   Entry
   23F91F50   CEEHDSPR       23BA0090     +000041B0   CEEHDSPR
   23F914E8                  23AB25E8     +0000005C   dllfunc
   23F91338   CEEVRONU       23CA3348     +00000706   CEEVRONU
   240316A0                  23AB13D0     +00000016   main
   24031720                  23CA1D10     +000009A4   CEEVROND
   23F910E0   EDCZHINV       23F64118     +0000009A   EDCZHINV
   23F91018   CEEBBEXT       00053380     +000001A6   CEEBBEXT
```

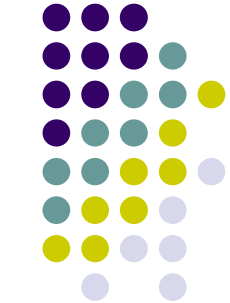
CEEDUMP Support for XPLink



31-bit

- After the Traceback, the DSAs on the stack are formatted.
- Individual DSAs labeled as:
  - "UPSTACK DSA" (Non-XPLink)
  - "DOWNSTACK DSA" (XPLink)
  - "TRANSITIONAL DSA" (LE "glue")
    - CEEVRONU -- RunOnUpstack
    - CEEVROND -- RunOnDownstack

CEEDUMP Support for XPLink

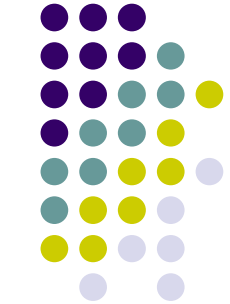


- Traceback support for **64-bit** stacks

```
Traceback:
  DSA       Entry       E  Offset  Load Mod  Program Unit  Service  Status
  00000001  CEEHDSP     +00000000  CELQLIB   CEEHDSP       HLE7709  Call
  00000002  CEEOSIGJ    +00000956  CELQLIB   CEEOSIGJ      HLE7709  Call
  00000003  CELQHROD    +00000256  CELQLIB   CELQHROD      HLE7709  Call
  00000004  CEEOSIGG    -17F50414  CELQLIB   CEEOSIGG      HLE7709  Call
  00000005  CELQHROD    +00000256  CELQLIB   CELQHROD      HLE7709  Call
  00000006  main        +000000CA  *PATHNAM                        Exception
  00000007  CELQINIT    +00001146  CELQLIB   CELQINIT      HLE7709  Call

  DSA       DSA Addr           E  Addr             PU Addr           PU Offset   Comp Date Attributes
  00000001  00000001082FABC0  00000000201C8F08    00000000201C8F08  00000000    20040312  XPLINK  EBCDIC  POSIX  Floating Point
  00000002  00000001082FD4E0  0000000020251100    0000000020251100  00000956    20040312  XPLINK  EBCDIC  POSIX  Floating Point
  00000003  00000001082FDEE0  0000000020182570    0000000020182570  00000256    20040312  XPLINK  EBCDIC  POSIX  Floating Point
  00000004  00000001082FE120  000000002024A7D8    000000002024A7D8  17F50408    20040312  XPLINK  EBCDIC  POSIX  Floating Point
  00000005  00000001082FEF40  0000000020182570    0000000020182570  00000256    20040312  XPLINK  EBCDIC  POSIX  Floating Point
  00000006  00000001082FF180  00000000200770C0    0000000000000000  ********    20040813  XPLINK  EBCDIC  POSIX  IEEE
  00000007  00000001082FF280  000000002013D010    000000002013D010  00001146    20040312  XPLINK  EBCDIC  POSIX  Floating Point
```

- Note that *execution* (RMODE) is still below-the-bar
- Only downstack transitions are from operating system
- Fewer control blocks than 31-bit; SYSMDUMP/IPCS VERBX LEDATA essential.

XPLink Transitions Tracing



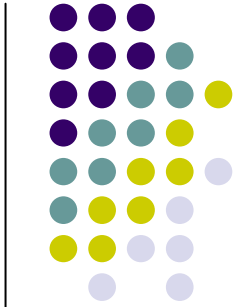
New for z/OS V1R8

- Trace level LE=20 for the TRACE run-time option specifies that transitions between XPLink and non-XPLink be recorded (31-bit only).
  - Trace entry type 7 occurs when an XPLINK function calls a non-XPLINK function.
  - Trace entry type 8 occurs when a non-XPLINK function calls an XPLINK function.
  - Format for trace table entries 7 and 8:

    ModuleNameOfCallingFunction : NameOfCallingFunction →

    ModuleNameOfCalledFunction : NameOfCalledFunction

XPLink Transitions Tracing

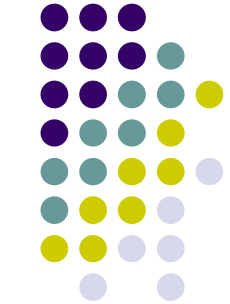


Example:

TRACE=(ON,32K,DUMP,LE=20)

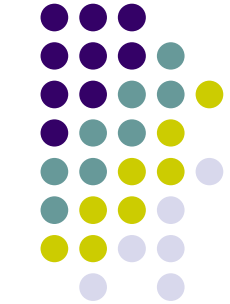
```
Displacement                 Trace Entry in Hexadecimal                                    Trace Entry in EBCDIC
-------    ------------------------------------------------------------------------  --------------------------------
+000000    Time 23.04.44.393727    Date 2005.04.29    Thread ID... 2050D5F000000000
+000010    Member ID.... 03    Flags..... 000000    Entry Type..... 00000008
+000018    C3C5D3C8 E5F0F0F3 40404040 40404040  7AC5C4C3 E9C8C9D5 E5404040 40404040  |CELHV003        :EDCZHINV       |
+000038    40404040 40404040 40404040 40404040  4060606E 81F8F5F9 83F0F1A7 40404040  |                  -->a859c01x    |
+000058    40404040 7A948189 95404040 40404040  40404040 40404040 40404040 40404040  |    :main                        |
+000078    40404040 404040F1                                                        |       1                         |

+000080    Time 23.04.44.399327    Date 2005.04.29    Thread ID... 2050D5F000000000
+000090    Member ID.... 03    Flags..... 000000    Entry Type..... 00000007
+000098    C3C5C5D7 D3D7D2C1 40404040 40404040  7AC3C5C5 D7C8E3D3 C3404040 40404040  |CEEPLPKA        :CEEPHTLC       |
+0000B8    40404040 40404040 40404040 40404040  4060606E C3C5C5D7 D3D7D2C1 40404040  |                  -->CEEPLPKA    |
+0000D8    40404040 7AC3C5C5 D7E3D3D6 D9404040  40404040 40404040 40404040 40404040  |    :CEEPTLOR                    |
+0000F8    40404040 404040F1                                                        |       1                         |

+000100    Time 23.04.44.425623    Date 2005.04.29    Thread ID... 2050D5F000000000
+000110    Member ID.... 03    Flags..... 000000    Entry Type..... 00000007
+000118    81F8F5F9 83F0F1A7 40404040 40404040  7A948189 95404040 40404040 40404040  |a859c01x        :main           |
+000138    40404040 40404040 40404040 40404040  4060606E 81F8F5F9 83F0F240 40404040  |                  -->a859c02     |
+000158    40404040 7A86A495 83F16D84 F1404040  40404040 40404040 40404040 40404040  |    :func1_d1                    |
+000178    40404040 404040F1                                                        |       1                         |

+000180    Time 23.04.44.427092    Date 2005.04.29    Thread ID... 2050D5F000000000
+000190    Member ID.... 03    Flags..... 000000    Entry Type..... 00000008
+000198    81F8F5F9 83F0F240 40404040 40404040  7A86A495 83F16D84 F1404040 40404040  |a859c02         :func1_d1       |
+0001B8    40404040 40404040 40404040 40404040  4060606E C3C5D3C8 E5F0F0F3 40404040  |                  -->CELHV003    |
+0001D8    40404040 7A979989 95A38640 40404040  40404040 40404040 40404040 40404040  |    :printf                      |
+0001F8    40404040 404040F1                                                        |       1                         |
```

Debugging an XPLink Application



- Major points of difference between XPLink and non-XPLink
  - The Stack
    - –upward-growing vs. downward-growing
    - –DSA format
    - –stack unwinding (backchain ptr vs. DSA size)
      - (BACKCHAIN suboption of XPLINK compile option)
    - –XPLink stack ptr (GPR4) is "biased" by 0x800 bytes

Debugging an XPLink Application

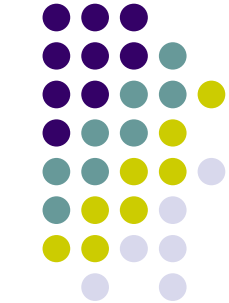


- Major points of difference between XPLink and non-XPLink continued...
  - Register conventions
    - Finding entry / return points, etc.
  - Parameter passing
    - R1 points to parm list vs. parms in regs and caller's DSA (STOREARGS suboption of XPLINK compile option)

Debugging an XPLink Application

- IPCS VERBX LEDATA
  - Similar support for tracebacks and DSAs as in CEEDUMP
- LE Storage Reporting
  - via RPTSTG Run-Time option
  - includes XPLink stack and threadstack statistics
- Full support for XPLINK-compiled functions provided by Debug Tool (**31-bit only**) and dbx debuggers
  - Through new CWIs provided by LE to traverse stack frames, locate entry points, etc.

Appendix F



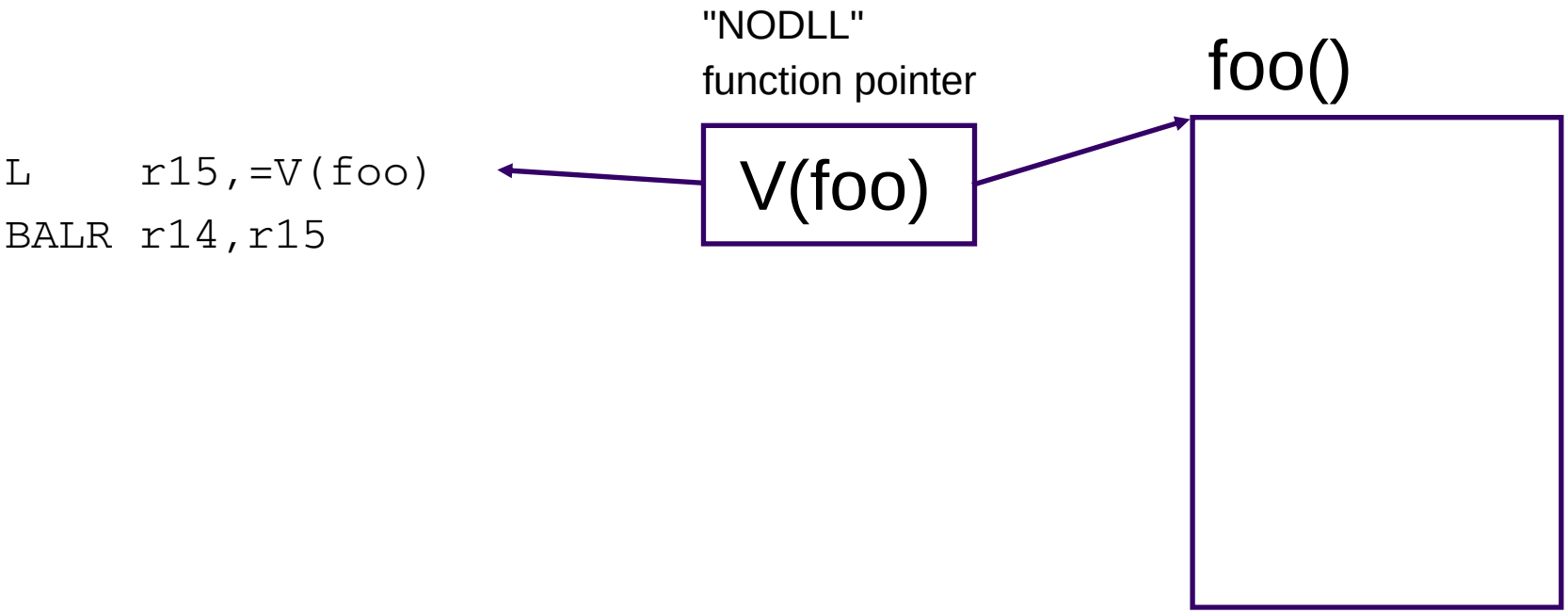
Callback Function Support

Reference Materials

Callback Function Support

Why is __bldxfd() needed?

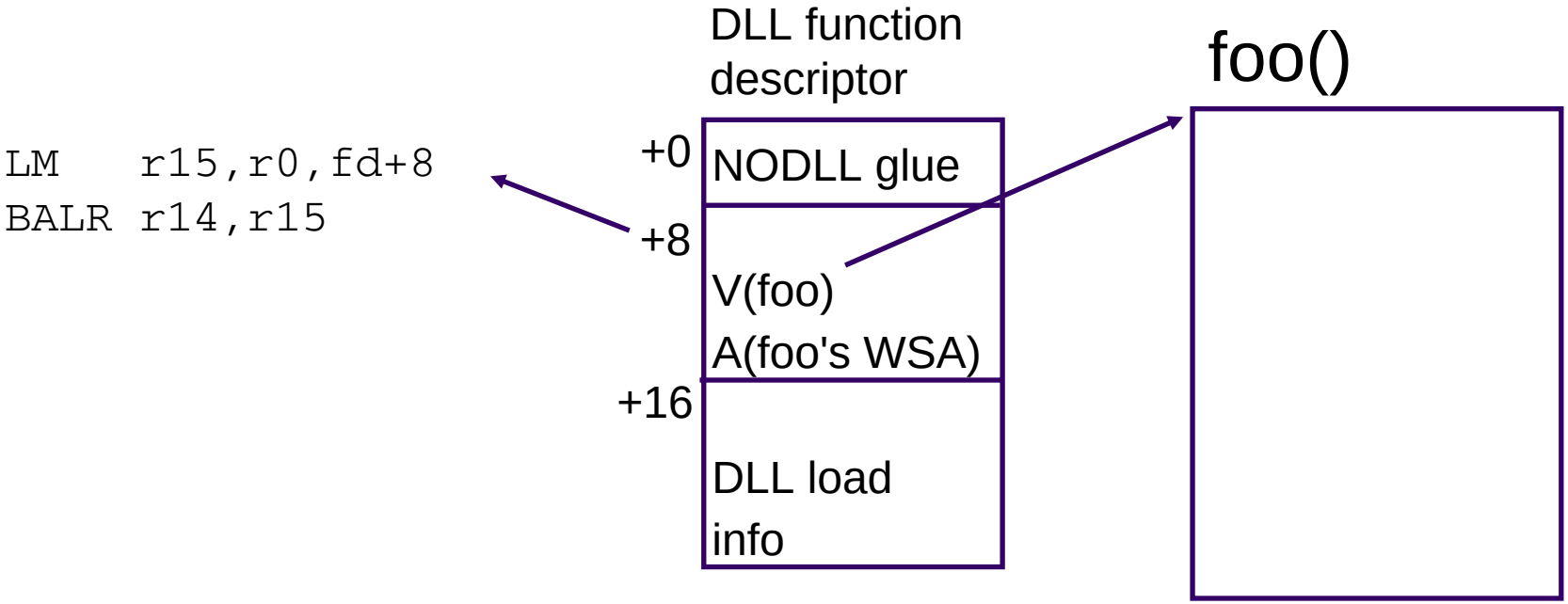
Taking the address of a function in NODLL-compiled code:



Callback Function Support...

Taking the address of a function in DLL-compiled code:

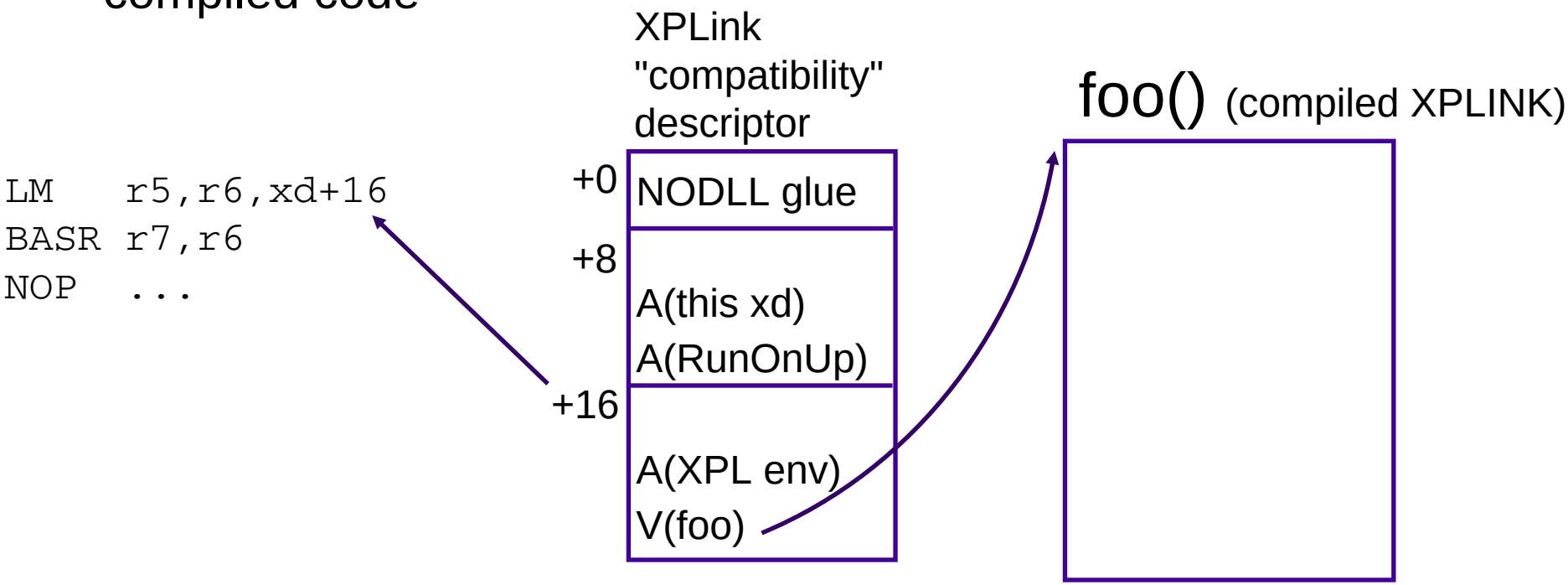
- A NODLL function pointer cannot be called from DLL-compiled code unless compiled DLL(CALLBACKANY)



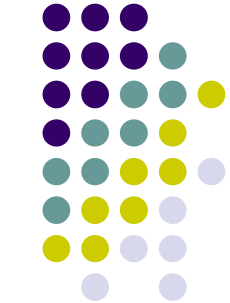
Callback Function Support...

Taking the address of a function in XPLINK-compiled code:

- A NODLL fp or DLL fd cannot normally be called from XPLINK-compiled code



Callback Function Support...



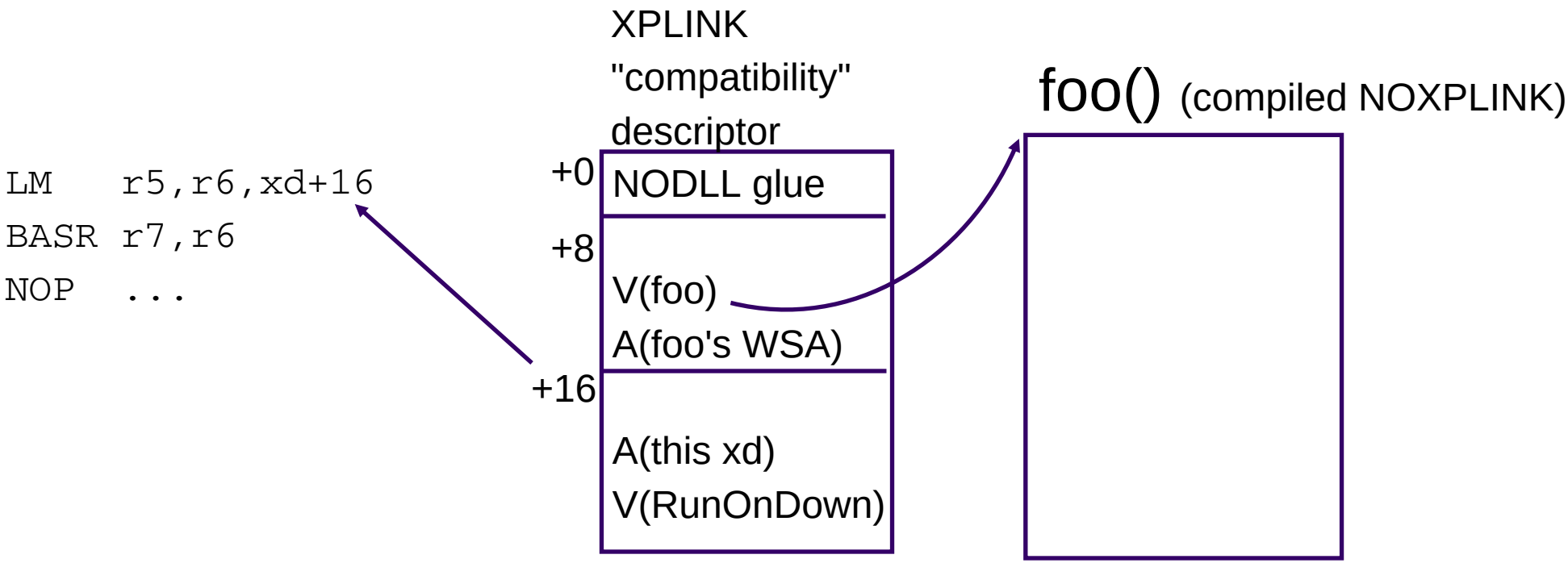
- A NODLL function pointer or DLL function descriptor cannot normally be called from XPLINK-compiled code
- The __bldxfd() CWI will interrogate the input "function pointer" and convert to an XPLink compatibility descriptor if necessary
- The __bldxfd() CWI will be called implicitly by the compiler for each function pointer parameter passed to an exported function

```
#pragma export(foo)
typedef int (*FP)(void);
void foo(FP fpParm1, int parm2, FP fpParm3) {
```

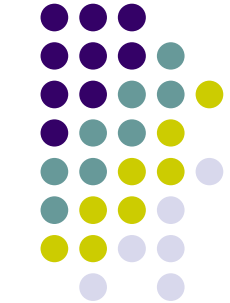
- __bldxfd() is *not* called for:
  - function pointers passed inside a structure, or global function pointers
- You need to call __bldxfd() explicitly in these cases, or see "New Compiler solutions to callback problem" in a couple of pages

Callback Function Support...

What if we take the address of NOXPLINK-compiled code from an XPLink function? Same rules apply, but different style XPLink compatibility descriptor.



New Compiler Solutions to Callback Problem

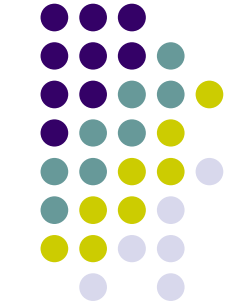


- __callback type cast qualifier
  - In the following example, all calls to (*func_p)() first result in a call to __bldxfd().

```
#if !__XPLINK_CALLBACK__
#define __callback
#endif

...
void (* __callback func_p) (void);
...
```

- XPLINK(CALLBACK) compiler option
  - *ALL* calls through function pointers result in a call to __bldxfd().
  - Not recommended for performance reasons.

64-bit solution to callback problem



Why again was __bldxfd() needed?

For taking the address of a function in NODLL-compiled code.

Why is __bldxfd() not needed in 64-bit?

Because there is no non-DLL non-XPLink code to interface with. All 64-bit function descriptors are RD-con/VD-con doubleword adcon pairs (16 bytes, loaded using LMG from zero past the descriptor location).