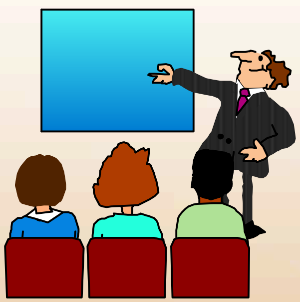
Converting your Language Environment C/C++ Applications to XPLINK Session 8121



Barry Lichtenstein
IBM Poughkeepsie
BarryL@us.ibm.com

Contents

- XPLINK Overview
- Choosing an XPLINK Application
- Building an XPLINK Application
- Running an XPLINK Application
- Debugging an XPLINK Application
- Appendices
- Performance Measurements
- Reference Materials

Background

- New workloads on OS/390 and z/OS consist largely of applications written on other platforms where function calls was "free"
- Some applications measure ~25% of execution time spent in function call overhead
- This is most serious in Object-Oriented applications, where functions tend to be smaller
  - that is, 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)
  - Reduce function footprint in memory
  - Provide a common linkage for C/C++ (and DLLs)
  - Provide compatibility with existing (non-XPLINK) code
  - No effect on existing applications

What's Happening with Today's Linkage?

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

Major Differences

- New register conventions
  - R13, R14 & R15 are just work registers
- New stack layout, grows to lower addresses
- No explicit check for overflow
  - Storage Protection mechanism used to detect stack overflow
- Improved parameter/return value passing

New Calling Convention

- Registers 6 and 7 used for linkage
  - formerly Registers 15 and 14
- Register 5 contains called function's own portion of WSA (its environment)
  - formerly Register 0 contained address of WSA, the called function had to compute the address of its own piece of Writeable Static
- No base register assumed on entry, call *may* have been made via Relative Branch
- Return register not preserved

New Prolog
Stack Overflow Detection

- No explicit test (mostly) for stack overflow
  - Functions with large stack frames still need explicit test
- 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

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)
- Arguments passed in caller's stack frame
  - can be addressed by caller using same base register as its own automatic storage
- Registers stored in called function's stack frame, not caller's

New Parameter Passing Conventions

- Argument area in fixed location in caller's stack
- Directly addressable by called function
- First 3 words passed in GPRs 1-3
- Up to 4 floating point arguments passed in FPRs
- Remaining arguments passed in storage

XPLINK Register Conventions

GPR 0	undefined	not preserved
GPR1	1st word of argument list or undefined	not preserved / 1st word of a returned aggregate
GPR 2	2nd word of argument list or undefined	not preserved / 2nd word of returned aggregate / high half of 64-bit integer return value
GPR 3	3rd word of argument list or undefined	not preserved / 3rd word of returned aggregate / 31-bit return value
GPR 4	Address of Stack Frame - 2048	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 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

Function Prolog Comparison

(page 58 or your handout)

"Old" prolog

```
000060  47F0  F022       B     34(,r15)
000064  01C3C5C5             CEE eyecatcher
000068  000000A0             DSA size
00006C  FFFFFFC0             =A(PPA1-f1)
000070  ... stack extension path
000082  90E4  D00C       STM
r14,r4,12(r13)
000086  58E0  D04C       L     r14,76(,r13)
00008A  4100  E0A0       LA    r0,160(,r14)
00008E  5500  C314       CL    r0,788(,r12)
000092  4130  F03A       LA    r3,58(,r15)
000096  4720  F014       BH    20(,r15)
00009A  58F0  C280       L     r15,640(,r12)
00009E  90F0  E048       STM
r15,r0,72(r14)
0000A2  9210  E000       MVI   0(r14),16
0000A6  50D0  E004       ST    r13,4(,r14)
0000AA  18DE             LR    r13,r14
```

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

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

XPLINK prolog

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

Choosing an XPLINK Application

- Highly-modular application with many calls to small functions is the ideal candidate
- XPLINK support provided in C/C++ Compiler
- In general, you cannot bind XPLINK-compiled and NOXPLINK-compiled functions together in the same program object
- Want to minimize calls between XPLINK and NOXPLINK-compiled functions (which requires expensive stack switching glue)
- Requires Binder, and the output executable must reside in a PDSE or the HFS

XPLINK Detractors

- The following can degrade performance or otherwise make using XPLINK unattractive:
  - 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
  - Using unsupported environment or function
  - Hex Math Library requires stack switch to run on Upstack (IEEE Floating Pt Library is ok, it's XPLINK)

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() 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)
  - Generates direct call using OS Linkage conventions (no glue, so fast) but only 72-byte savearea (e.g.. C headers)
- #pragma linkage(..., OS_UPSTACK)
  - 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...

- 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
- DB2 stored procedures (EXEC SQL is allowed)
- IMS transactions (calls to ctdli() are allowed)
- ~~PIPI~~ (added 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

Building an XPLINK Application

	Today...	For XPLINK...
SYSLIB static libraries	SCEELKED SCEELKEX SCEEOBJ SCEECPP	SCEEBND2 ~~SCEEBIND~~
Dynamic Link Library (DLL) side decks	None for LE	In SCEELIB: ▸ CELHS003 (C RTL) ▸ CELHS001 (LE AWIs) ▸ CELHSCPP (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

XPLINK Compile Option

NOXPLINK | XPLINK (optional suboptions)

- XPLINK(BACKCHAIN | NOBACKCHAIN)
  - 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)
  - Compiler inserts extra code after prolog to explicitly store parameter registers into argument area in caller's DSA.

XPLINK Compile Option...

NOXPLINK | XPLINK (optional suboptions)

- XPLINK(OSCALL(Downstack | Upstack | <u>Nostack</u>))
  - Alters default behavior of `#pragma linkage(..., OS)`
- XPLINK(NOGUARD | <u>GUARD</u>)
  - NOGUARD will generate an explicit check of the stack floor in the prolog code
- XPLINK(<u>NOCALLBACK</u> | CALLBACK) z/OS R5
  - CALLBACK will allow non-XPLink function pointers or descriptors to be correctly used in an XPLink program. __callaback qualifier is preferred.

c89 changes

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

c89 simple example

- XPLINK "Hello World" example:

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

Running an XPLINK Application

- Requires that both LE run-time libraries are available at execution time:
  - SCEERUN
  - SCEERUN2
    - New
    - It's a PDSE (required by XPLINK)
    - Contains XPLINK versions of C RTL, locales and converters, more

New LE Run-Time Option

- XPLINK(ON|<u>OFF</u>)
  - 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)

Changed LE Run-Time Options

- 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

New/Changed LE Run-Time Options

- 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**

New/Changed LE Run-Time Options...

- 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)

Changed LE Run-Time Options

- 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

Debugging an XPLINK Application

CEEDUMP support for XPLINK

- Traceback support for Up and Down stacks

```
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
```

Debugging an XPLINK Application

CEEDUMP support for XPLINK

- 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

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 (see Appendix A)
    - 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 and dbx debuggers
  - Through new CWIs provided by LE to traverse stack frames, locate entry points, etc.

New CWIs (Compiler Writer Interfaces)

- 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
  - __ep_find()
    - Takes as input a DSA address and format
    - Returns the address of the entry point of the function owning the input DSA

New CWIs (Compiler Writer Interfaces)...

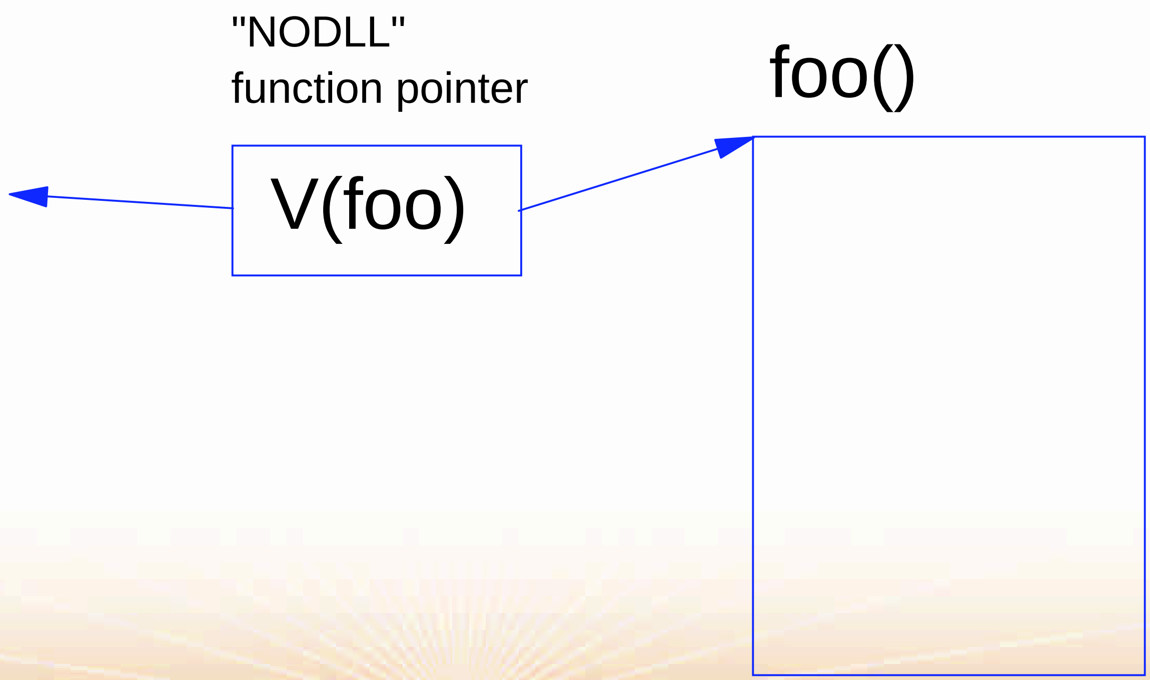
- __bldxfd()
  - XPLINK environment only
  - Takes a function pointer as input
  - Returns a "function pointer" that can be called by all linkage types

Callback function support

Why is __bldxfd() needed?

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

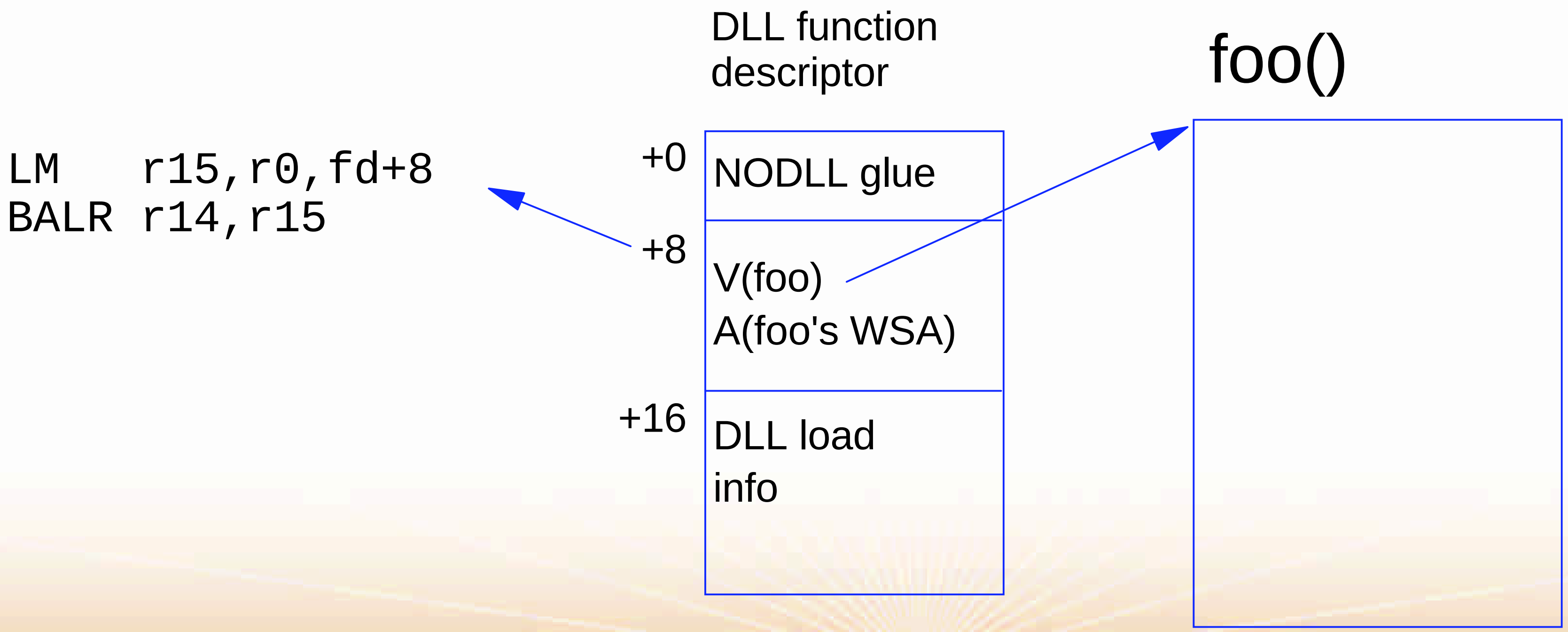
```
L    r15,=V(foo)
BALR r14,r15
```



Callback function support...

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

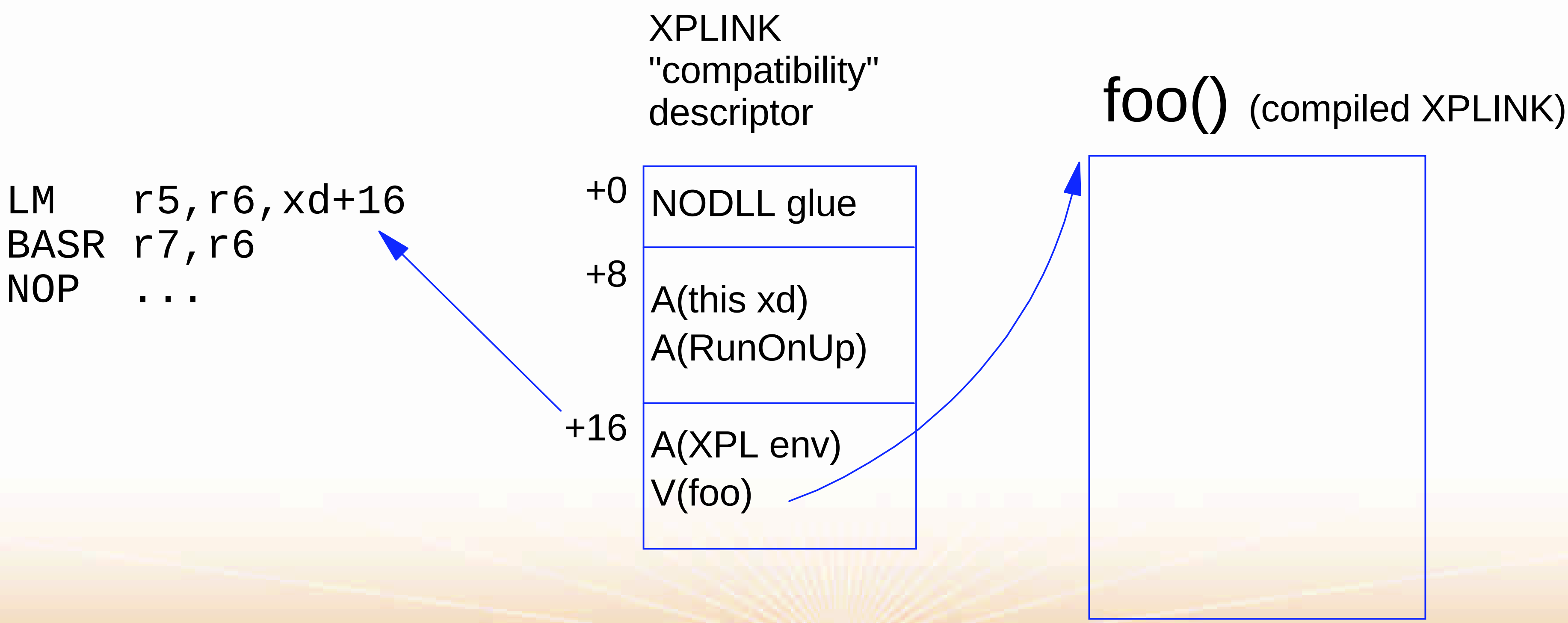
- A NODLL function pointer cannot be called from DLL-compiled code unless compiled DLL(CALLBACKANY)



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...

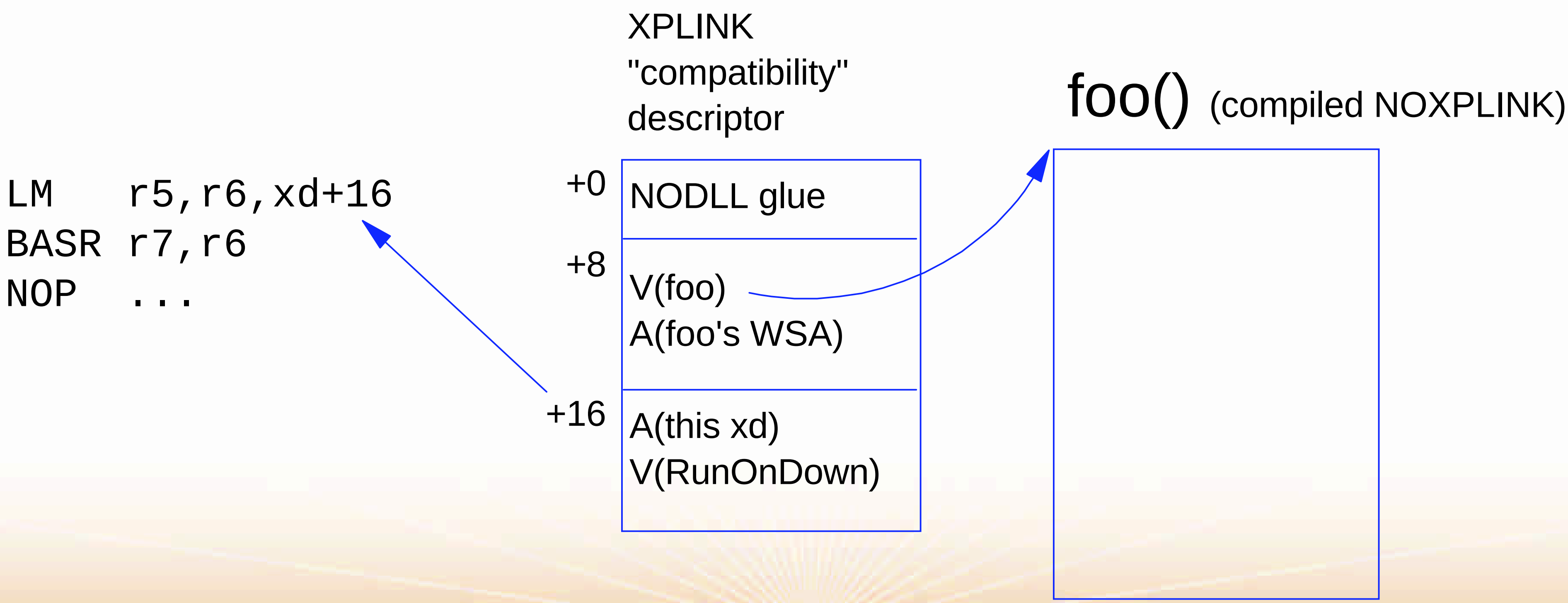
- A NODLL function pointer or DLL function descriptor cannot normally be called from XPLINK-compiled code
- The __bldxfd() CWI will interrogate the input "function pointer" and convert to an XPLINK compatibility descriptor if necessary
- The __bldxfd() CWI will be called implicitly by the compiler for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

Appendix A
Comparison of Register Conventions

	Current	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
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 stk frame size

Appendix B
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 C

Main XPLINK Publications of Interest

- LE Programming Guide (SA22-7561) has a chapter on developing XPLINK applications.
- 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 Redbook (SG24-5991, ibm.com/redbooks)
- C/C++ for z/OS books updated too

Performance Measurements

Reference Materials

XPLINK Performance Redbook

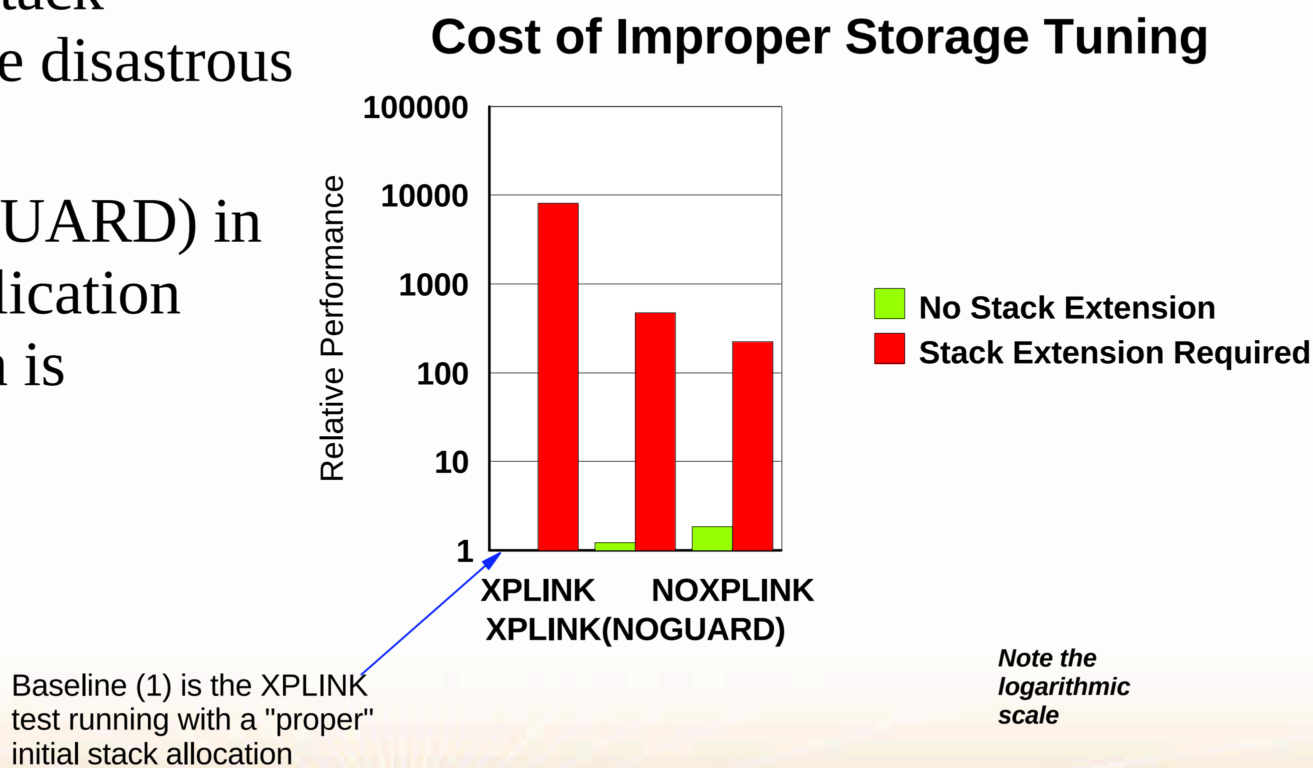
- Measurements made over summer 2000
- SG24-5991

  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

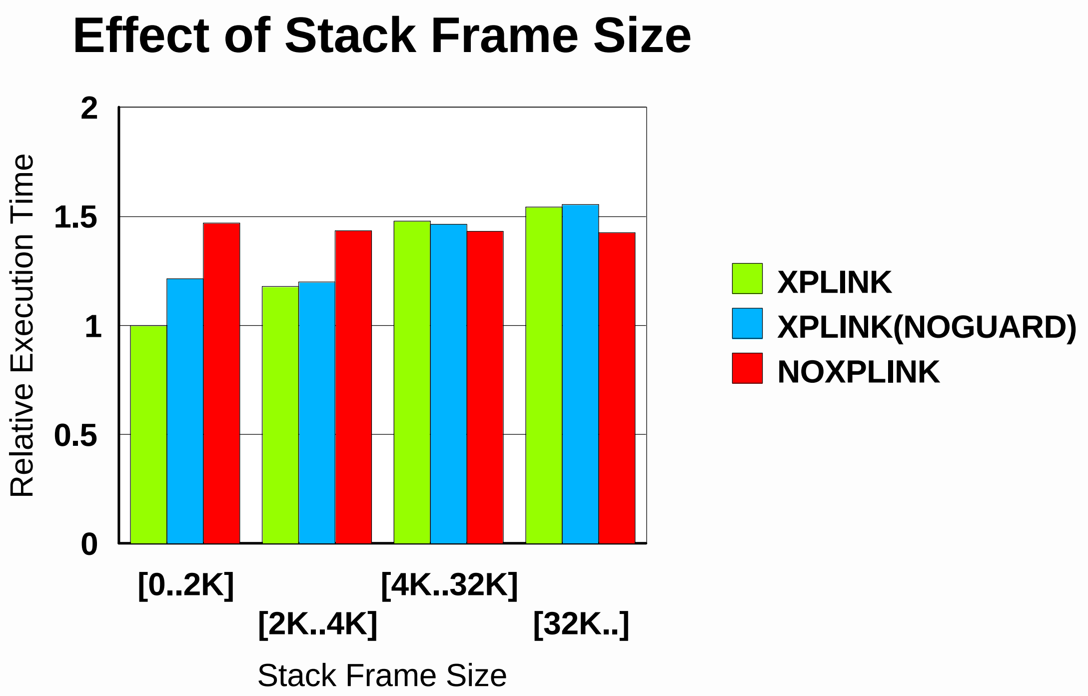
The Importance of Storage Tuning

- 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



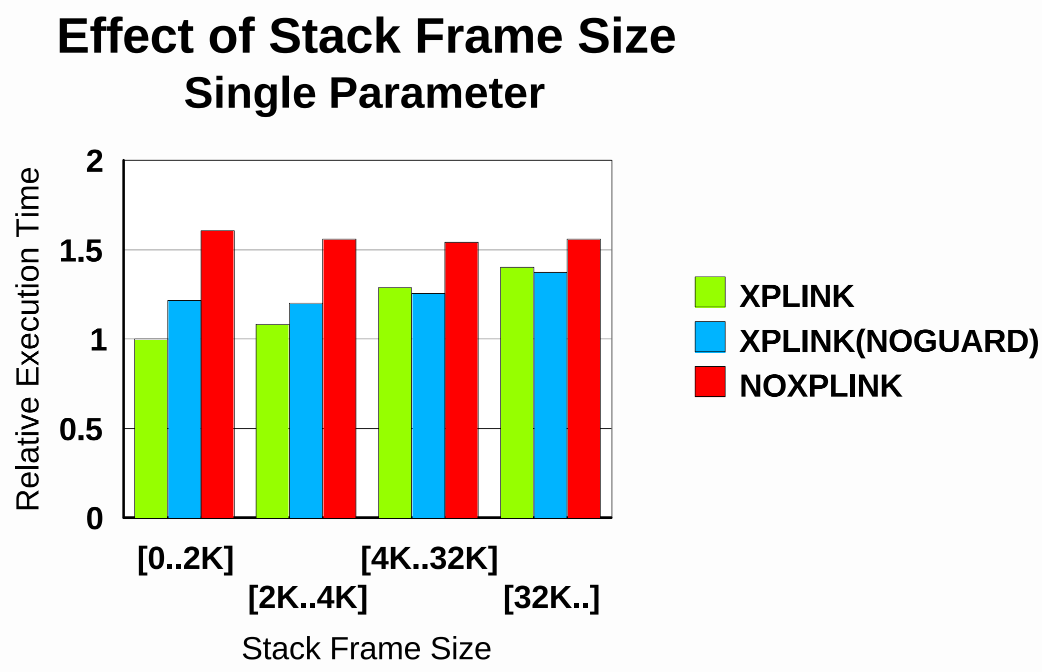
Effect of Stack Frame Size

- XPLINK is optimized for small stack frames (that is, small amounts of automatic storage)
  - We expect the slower cases to get better in the future



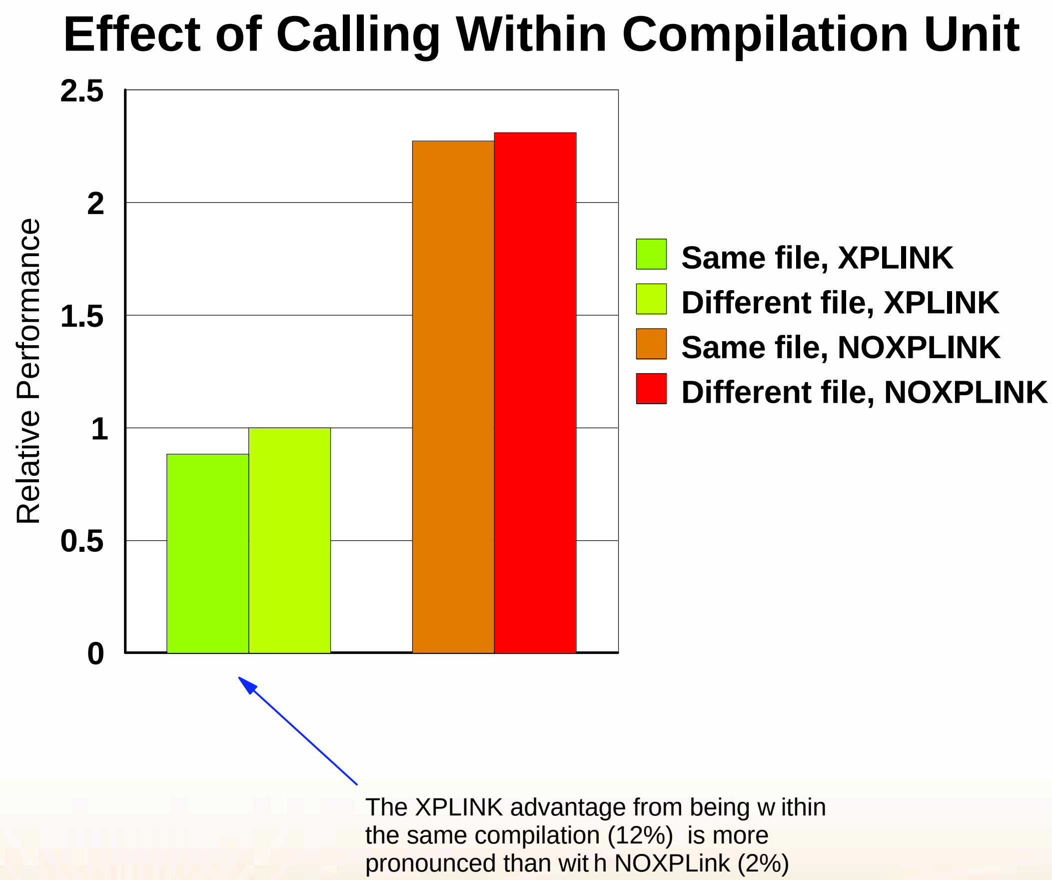
Effect of Number of Parameters

- XPLINK function 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

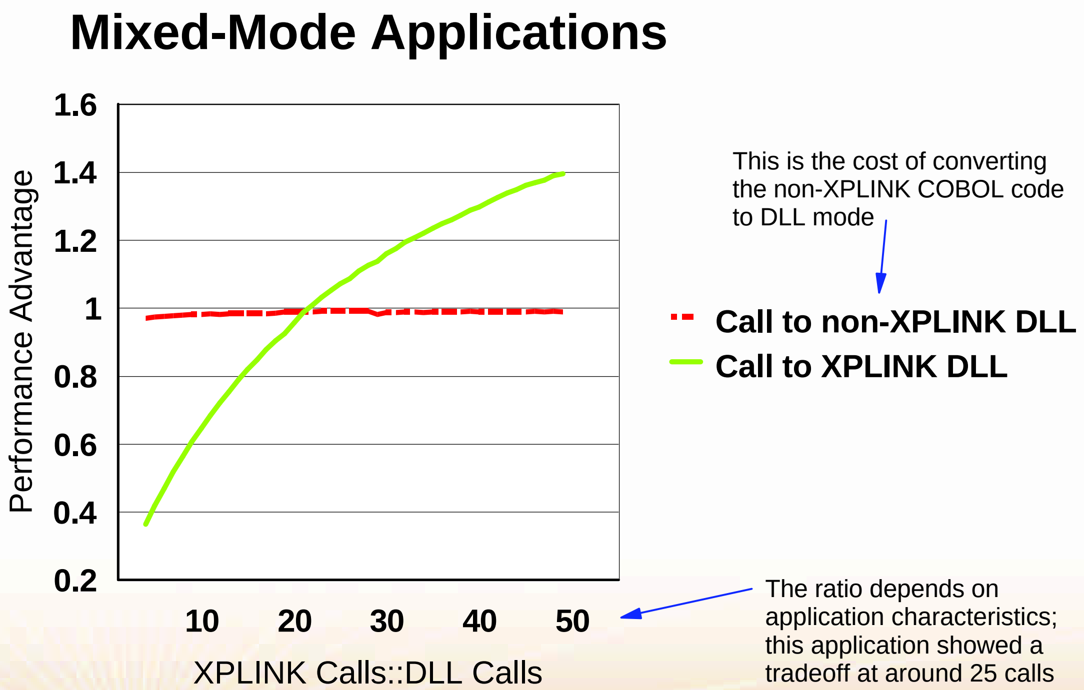


Mixing XPLINK with a COBOL Application

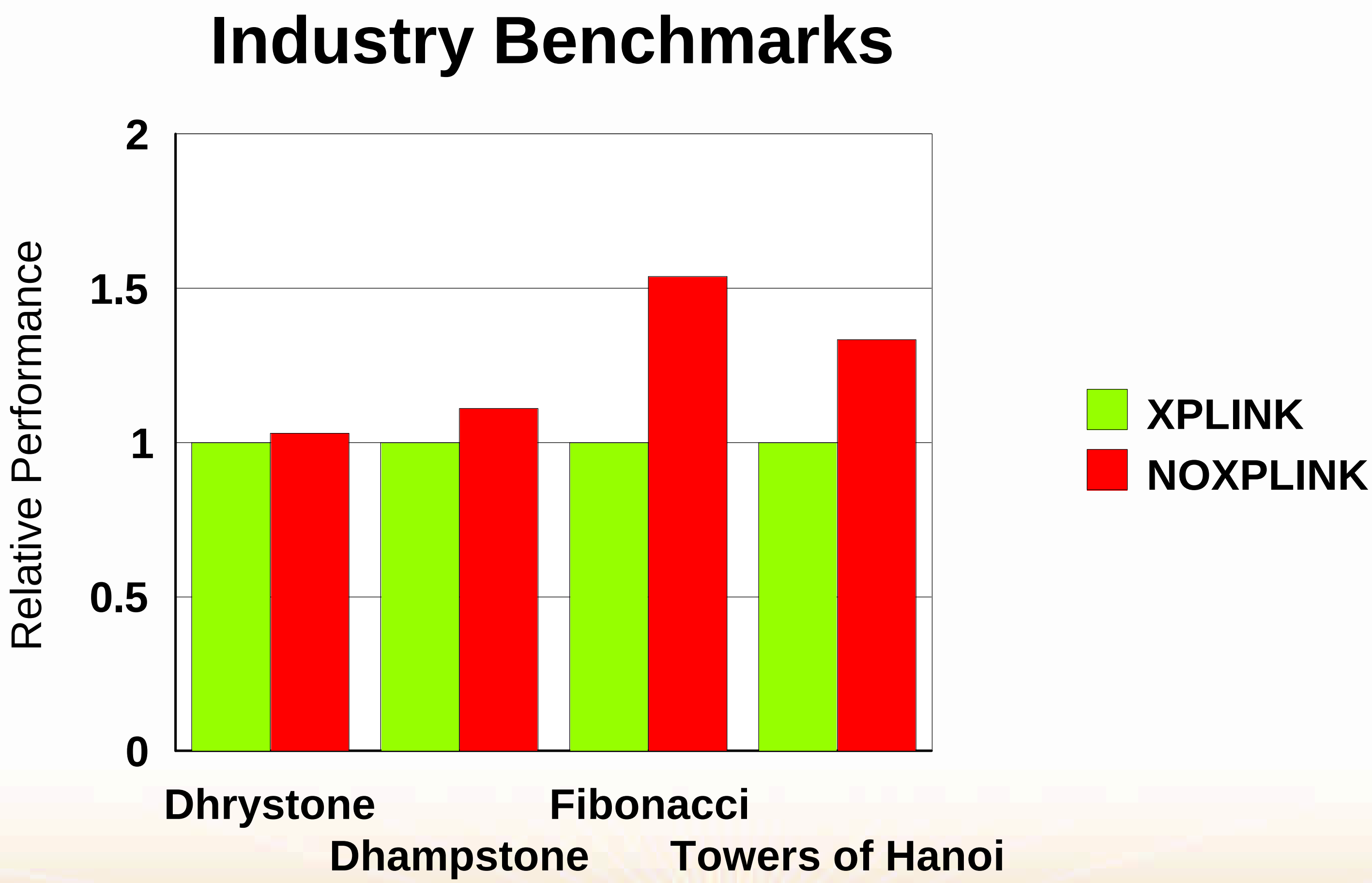
- COBOL does not support XPLINK
- Separate the XPLINK (C/C++) code from the non-XPLINK code
- Put XPLINK code into a DLL

Mixing XPLINK with a COBOL Application

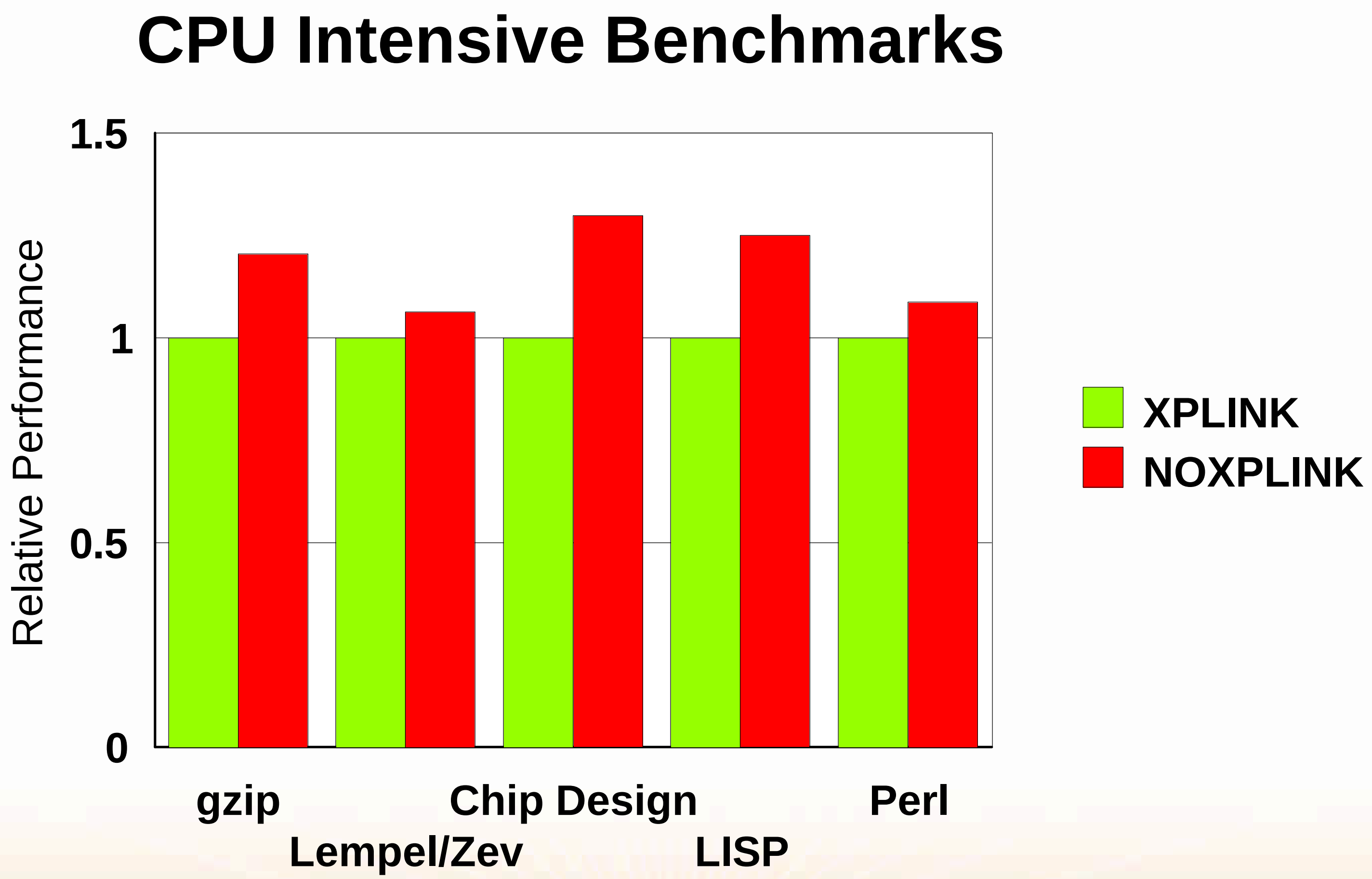
- Overall application performance depends on the number of calls ***inside*** the DLL for every call ***into*** the DLL
- This is typical of the performance characteristics expected from a C/C++ DLL written for use with a COBOL DLL.



Industry Benchmarks



CPU Intensive Benchmarks



XPLINK
The Details

Reference Materials

Function Prolog Comparison

"Old" prolog

```
000060  47F0  F022       B    34(,r15)
000064  01C3C5C5              CEE eyecatcher
000068  000000A0              DSA size
00006C  FFFFFFC0              =A(PPA1-f1)
000070  ... stack extension path
000082  90E4  D00C       STM  r14,r4,12(r13)
000086  58E0  D04C       L    r14,76(,r13)
00008A  4100  E0A0       LA   r0,160(,r14)
00008E  5500  C314       CL   r0,788(,r12)
000092  4130  F03A       LA   r3,58(,r15)
000096  4720  F014       BH   20(,r15)
00009A  58F0  C280       L    r15,640(,r12)
00009E  90F0  E048       STM  r15,r0,72(r14)
0000A2  9210  E000       MVI  0(r14),16
0000A6  50D0  E004       ST   r13,4(,r14)
0000AA  18DE             LR   r13,r14
```

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

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

XPLINK prolog

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

Where Have all the Instructions Gone?

- better register conventions reduce number of registers saved (2 vs 6 here)
- moving control information out of line

```
0060  47F0  F022       B      34(,r15)
0064  01C3C5C5                CEE eyecatcher
0068  000000A0                DSA size
006C  FFFFFFC0                =A(PPA1-f1)
0070  ... stack extension path
0082  90E4  D00C       STM    r14,r4,12(r13)
0086  58E0  D04C       L      r14,76(,r13)
008A  4100  E0A0       LA     r0,160(,r14)
008E  5500  C314       CL     r0,788(,r12)
0092  4130  F03A       LA     r3,58(,r15)
0096  4720  F014       BH     20(,r15)
009A  58F0  C280       L      r15,640(,r12)
009E  90F0  E048       STM    r15,r0,72(r14)
00A2  9210  E000       MVI    0(r14),16
00A6  50D0  E004       ST     r13,4(,r14)
00AA  18DE             LR     r13,r14
```

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

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

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

```
0048  9067  4788       STM
r6,r7,1928(r4)
004C  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)
```

- detecting overflow with guard page
- 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)

Prolog Comparison for Large Automatic Storage

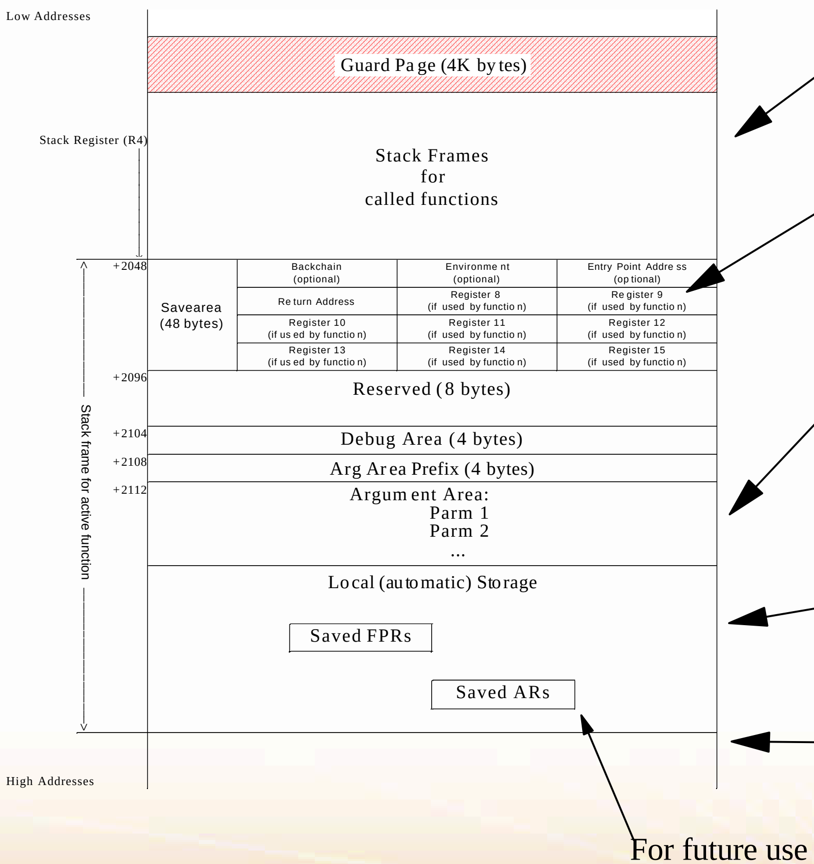
- Functions with large automatic storage clearly do not get the same performance advantage with XPLINK

```
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
   // ...
   //
   // ...
};
```

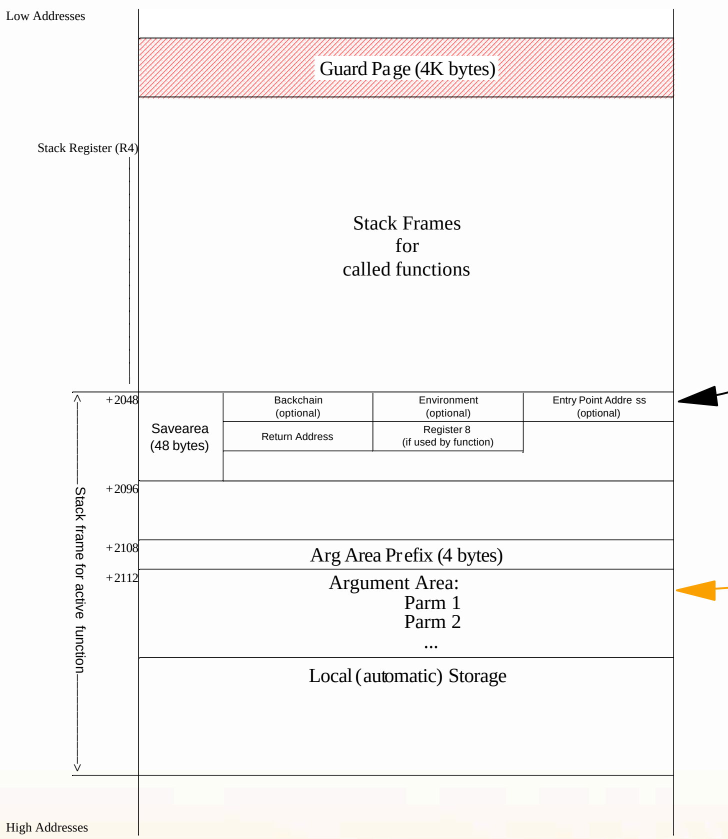
```
000048  9023  4844        STM    r2,r3,2116(r4)
00004C  1804              LR     r0,r4
00004E  0D20              BASR   r2,0
000050  A72A  0034        AHI    r2,H'52'
000054  5A40  2000        A      r4,0(,r2)
000058  5940  C364        C      r4,868(,r12)
00005C  A744  0022        JL      stack extender
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



- 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

Calling (1)

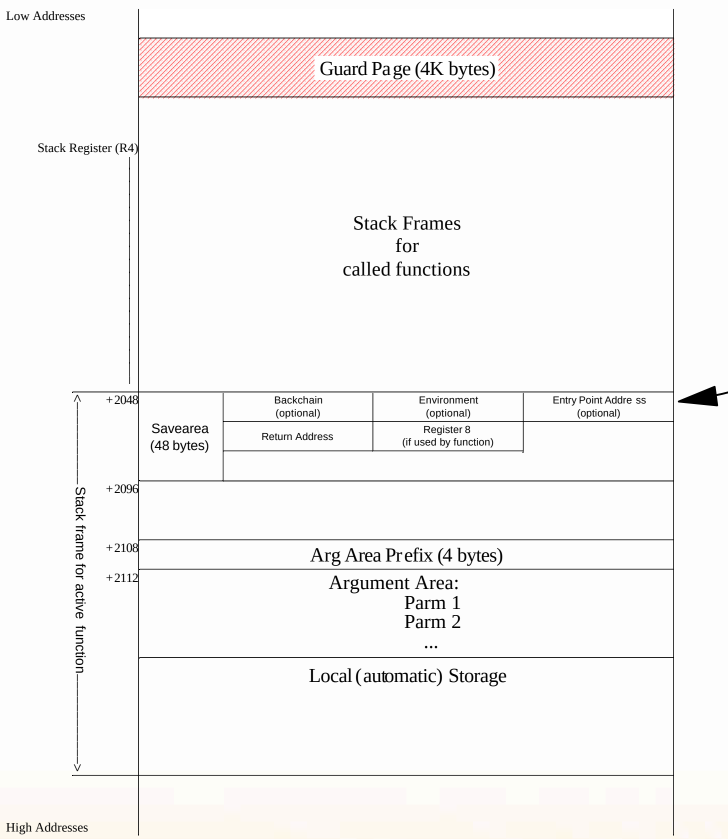



```
void f1(void) {
   f2();
      nop   flag
};

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

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

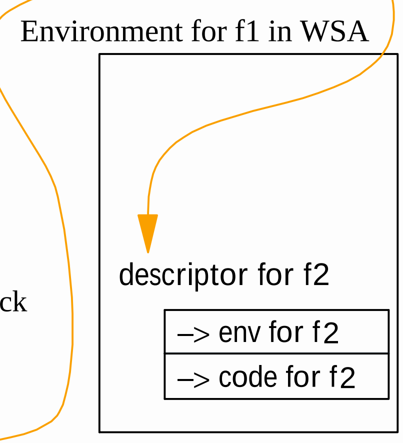
   some arguments are passed in registers; more about this later

Calling (2)




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

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

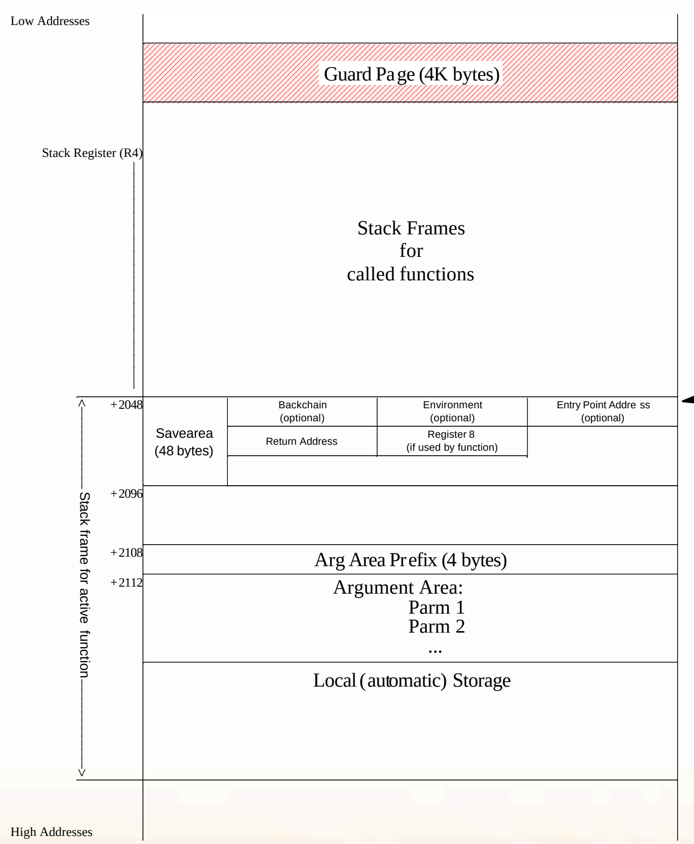
```
000050  9856  5000      LM    r5,r6,=A(f2)(r5,0)
000054  0D76            BASR  r7,r6
000056  4700  FFFD      NOP   4093(,r15)
```

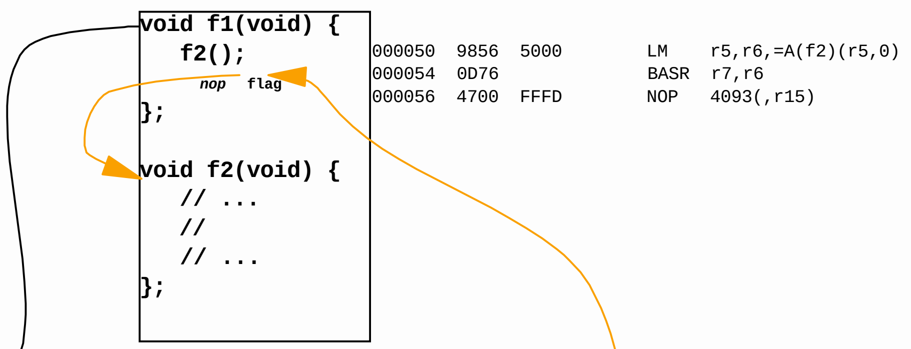


1. Construct arguments in current stack frame's argument area
2. Pick up addresses of function's "environment" and code from the function descriptor

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

Calling (3)

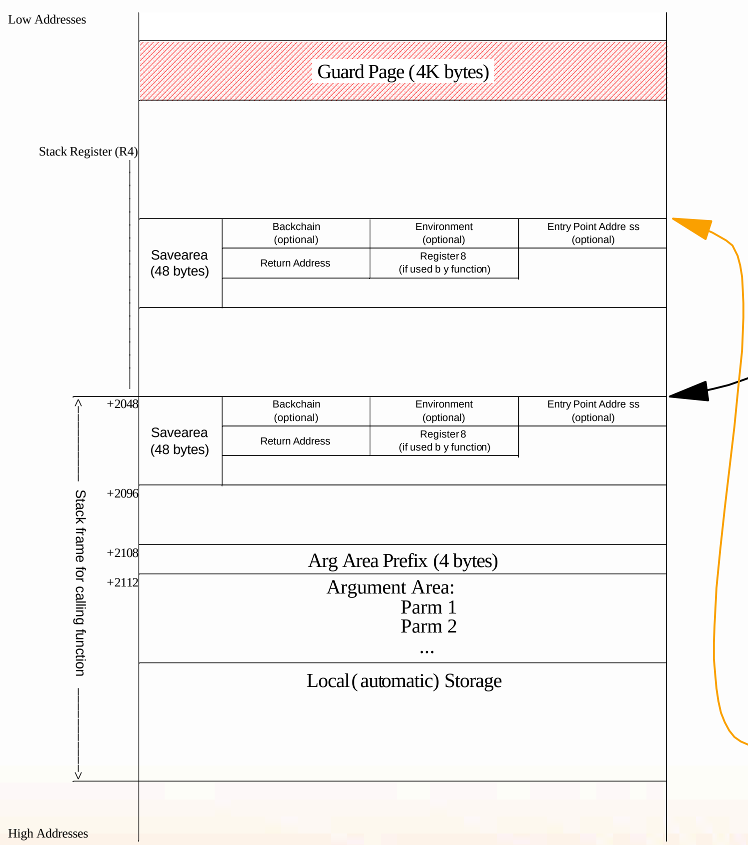




1. Construct arguments in current stack frame's argument area
2. Pick up addresses of function's "environment" and code from the function descriptor
3. Link (BASR or BRAS) to called function

   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 initialised by the Binder on detecting that the called function is imported

Calling (4)



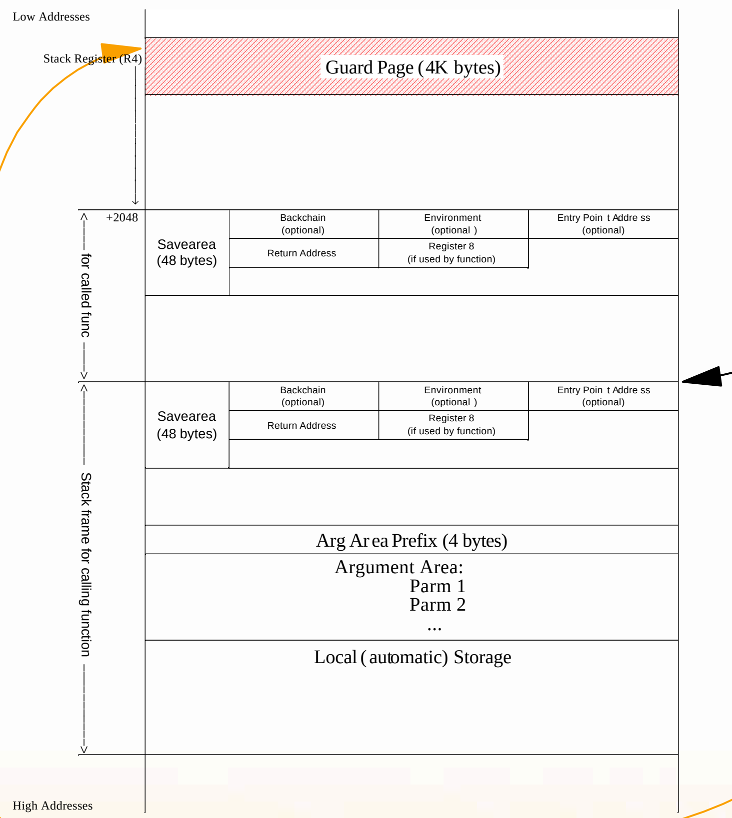

```
void f1(void) {
   f2();
     nop   flag
};

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

Calling (5)



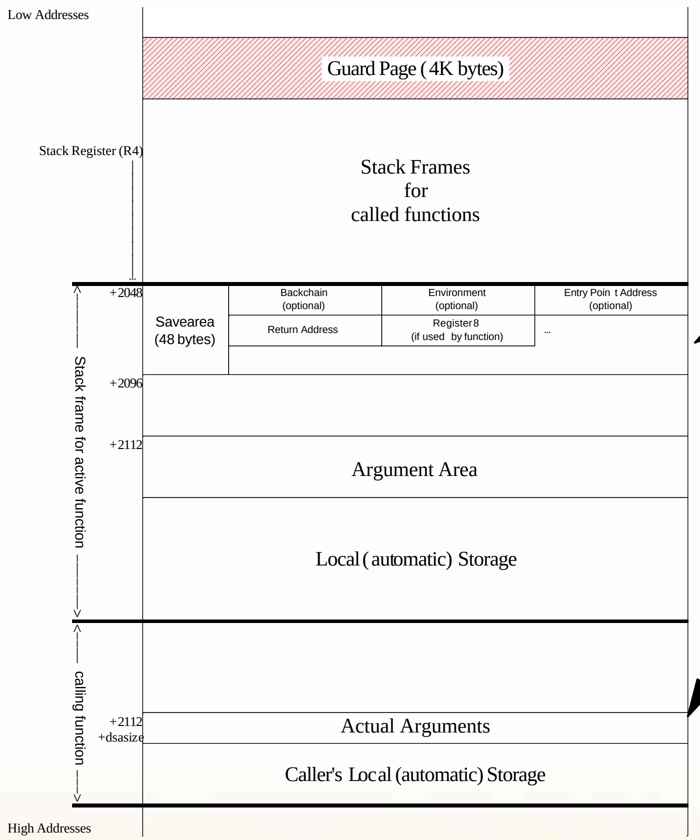

```
void f1(void) {
   f2();
     nop   flag
};

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

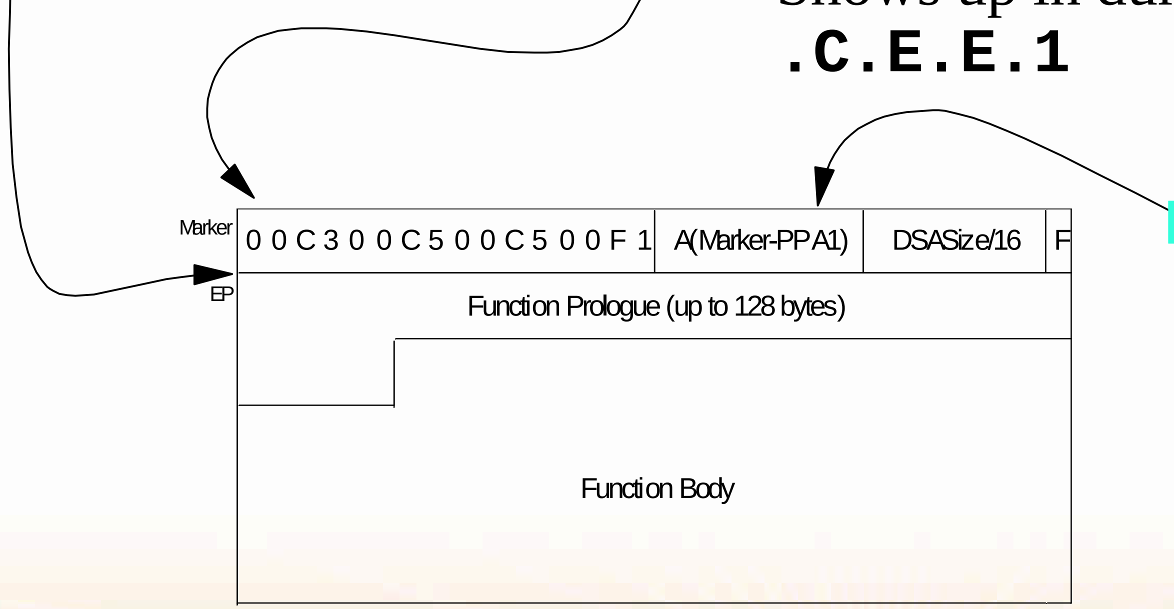
Argument Addressability



- Active Stack Frame
- Actual Arguments to Active function, built by caller in caller's argument area. Addressable at (R4+2112+*active stack frame size*)

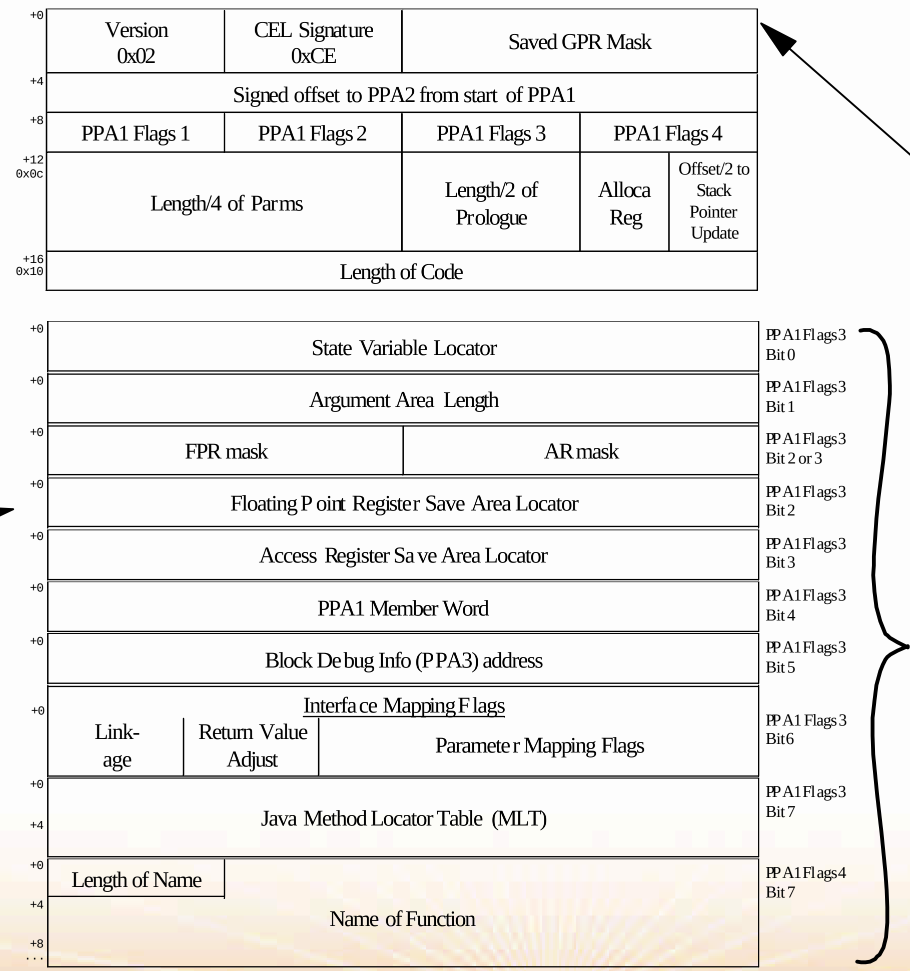
Entry Point Marker

- 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

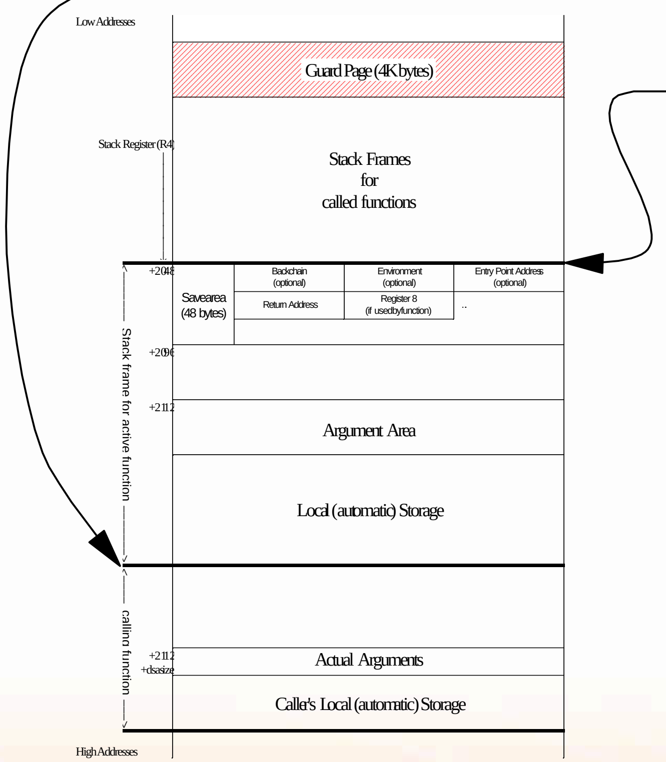


"Locators" are 4-bit register numbers followed by 28-bit offsets. Add the contents of the specified register and the specified offset to get to the target of the locator

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

Stack Walking

The Stack:



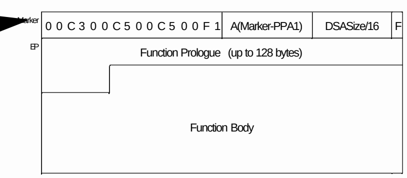

```
void f1(void) {
   f2();
     nop  flag
};

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

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

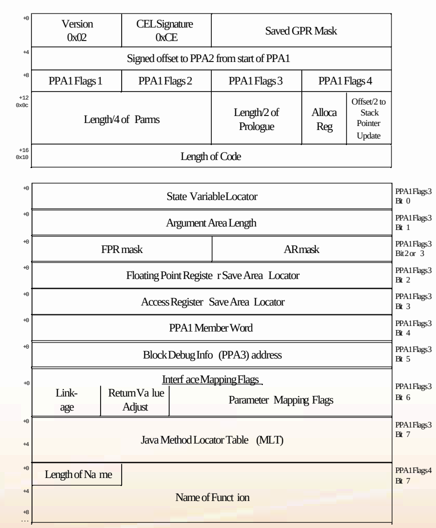
The steps described here assume we are ***not*** in f2()'s prolog. We can determine this by:

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

The Code:

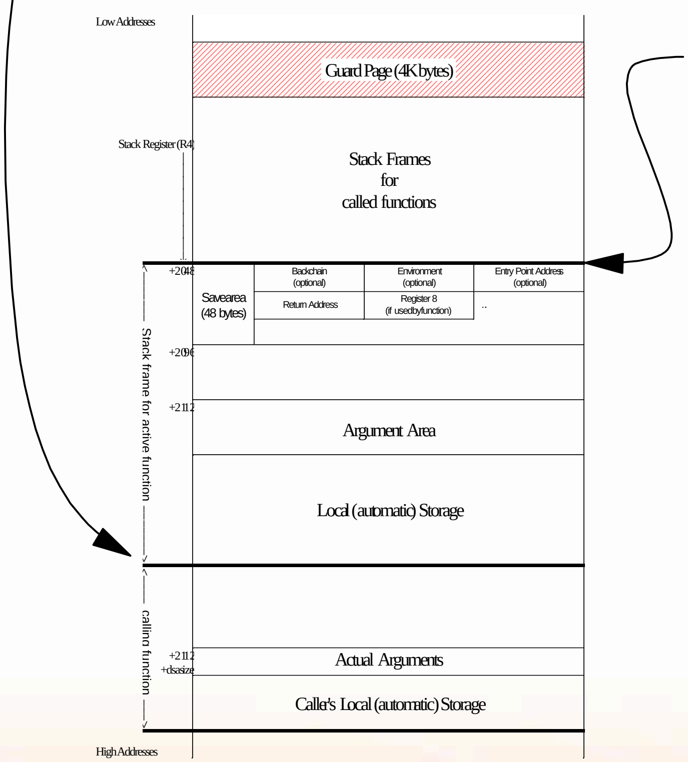


Static Data:



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 fram e 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 an 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 interrup t; 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 PPA 1 and add the dsasize to that register
9. Repeat as required

Stack Walking Steps (continued)

- Stack structure is fully supported by
  - Debug Tool
  - 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 Descriptors

- NOP at call site
- 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
0001	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

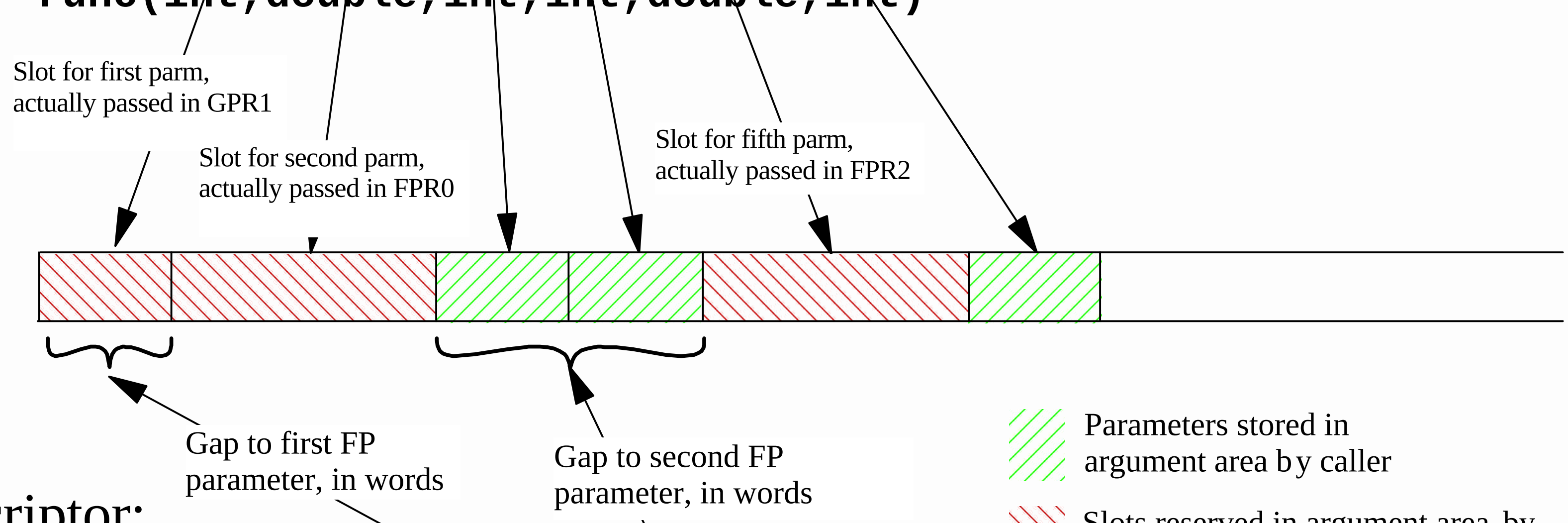
Call Descriptor...

- 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

```
func(int,double,int,int,double,int)
```



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

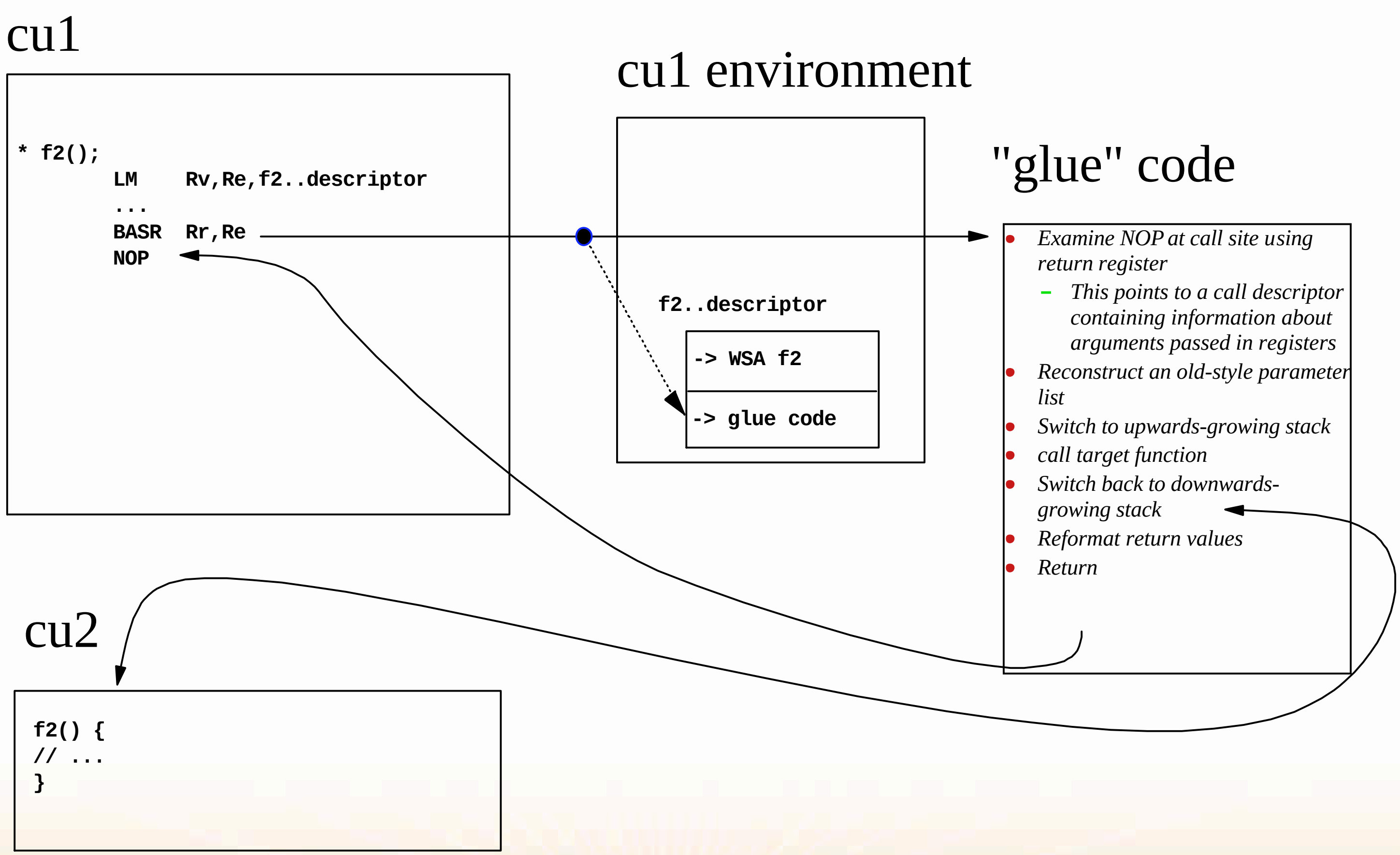
Examining Actual Arguments

- 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



Compatibility: Old Calling New