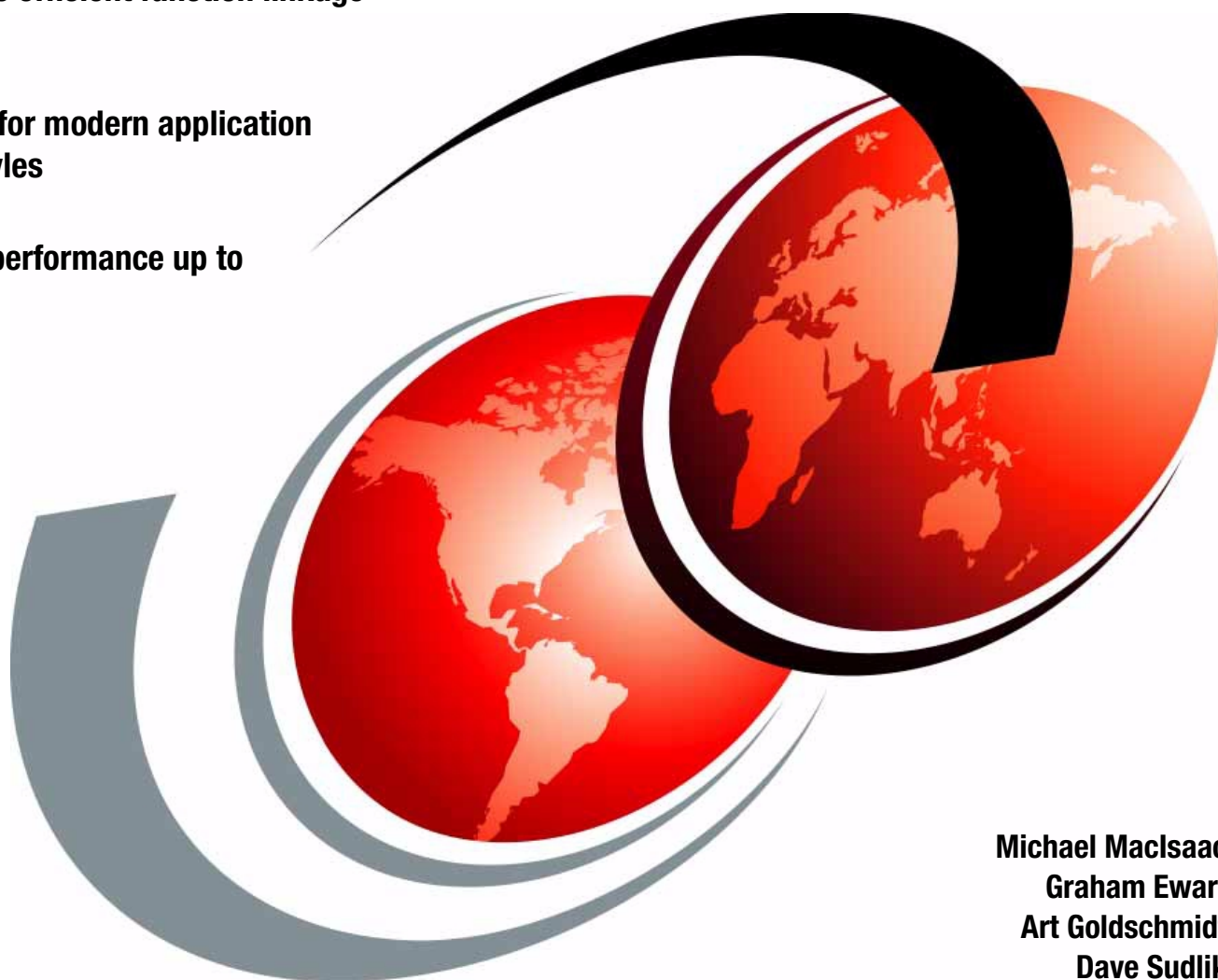


# XPLink: OS/390 Extra Performance Linkage

New, more efficient function linkage

Designed for modern application  
coding styles

Increase performance up to  
33%



Michael MacIsaac  
Graham Ewart  
Art Goldschmidt  
Dave Sudlik





International Technical Support Organization

SG24-5991-00

**XPLink:  
OS/390 Extra  
Performance Linkage**

December 2000

**Take Note!**

Before using this information and the product it supports, be sure to read the general information in Appendix C, "Special notices" on page 77.

**First Edition (December 2000)**

This edition applies to Version 2, Releases 10 and later of OS/390 C/C++ (program number 5647-A01).

Comments may be addressed to:  
IBM Corporation, International Technical Support Organization  
Dept. HYJ Mail Station P099  
2455 South Road  
Poughkeepsie, NY 12601-5400

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

**© Copyright International Business Machines Corporation 2000. All rights reserved.**

Note to U.S. Government Users - Documentation related to restricted rights - Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

---

# Contents

<b>Preface</b> .....	v
The team that wrote this redbook .....	v
Comments welcome .....	vi
 <b>Chapter 1. Overview</b> .....	1
 <b>Chapter 2. Technical overview</b> .....	3
2.1 History .....	3
2.2 What is new .....	5
2.3 Performance benefits of the new calling conventions .....	5
2.3.1 Stack organization .....	5
2.3.2 Stack layout .....	8
2.3.3 Argument passing conventions .....	9
2.4 Future of non-XPLink .....	10
 <b>Chapter 3. Terminology and usage</b> .....	13
3.1 Installation and customization .....	13
3.2 Compiling and binding an XPLink application .....	15
3.2.1 Compiler support .....	15
3.2.2 Binder support .....	16
3.2.3 Building XPLink applications from a UNIX shell .....	17
3.2.4 Building XPLink applications with JCL .....	18
3.2.5 Building XPLink applications under TSO .....	18
3.3 Running an XPLink application .....	18
3.4 Debugging an XPLink application .....	20
3.4.1 Debugger support .....	20
3.5 Tracing an XPLink application .....	20
3.6 Storage tuning .....	21
3.6.1 Recompile to determine stack requirements .....	21
3.6.2 Estimate stack requirements .....	21
3.6.3 Unknowable stack requirements .....	22
 <b>Chapter 4. Compatibility considerations</b> .....	23
4.1 Mixing XPLink and non-XPLink code .....	23
4.2 Mixing assembler code with XPLink .....	24
4.2.1 XPLink assembler .....	24
4.2.2 Non-XPLink assembler .....	28
4.3 Callback function considerations .....	32
4.4 XPLink restrictions .....	34
4.5 OS/390 C/C++ Version 2 Release 10 enhancements .....	35
4.5.1 Generalized Object File Format (GOFF) .....	35
4.5.2 InterProcedural Analysis (IPA) level 2 .....	36
4.5.3 Addition of @STATIC map into compiler listing .....	36
4.5.4 COMPACT compiler option .....	36
4.6 IPA and XPLink .....	37
 <b>Chapter 5. Performance characteristics</b> .....	39
5.1 Improvements in function prologs .....	39
5.2 The effect of stack size on XPLink performance .....	40
5.2.1 The effect of parameter-list size .....	42
5.2.2 The effect of calling within a compilation unit .....	43

5.3	The effect of XPLink suboptions on performance . . . . .	44
5.4	The cost of glue code . . . . .	45
5.5	The cost of a mixed XPLink-assembler application . . . . .	46
5.6	The cost of calling the C run-time library . . . . .	49
5.7	XPLink and object-oriented code . . . . .	50
5.8	XPLink and C++ exception handling . . . . .	50
<b>Chapter 6.</b>	<b>Application and benchmark measurements . . . . .</b>	<b>53</b>
6.1	Quantification of the value of a system . . . . .	53
6.1.1	Total cost of ownership . . . . .	54
6.1.2	Minimization of Language Environment path length . . . . .	54
6.2	Benchmarks . . . . .	55
6.2.1	CPU instruction-intensive development benchmarks . . . . .	55
6.2.2	C run-time library . . . . .	56
6.2.3	Industry standard compute-intensive benchmarks . . . . .	59
6.2.4	IBM internal benchmarks . . . . .	61
6.2.5	Compression application . . . . .	62
6.2.6	The Monte Carlo application . . . . .	63
6.3	Middleware benchmarks . . . . .	64
6.3.1	Lotus Domino . . . . .	64
6.3.2	SAP R/3. . . . .	65
6.3.3	Component Broker. . . . .	65
6.4	Solution developer and customer client/server applications . . . . .	66
6.4.1	Measurement context . . . . .	66
6.4.2	net.TABLES from Data Kinetics Ltd. . . . .	66
<b>Appendix A.</b>	<b>XPLink code sequences . . . . .</b>	<b>73</b>
A.1	Code sequences for function prologs. . . . .	73
<b>Appendix B.</b>	<b>Sample CEEDUMP header. . . . .</b>	<b>75</b>
<b>Appendix C.</b>	<b>Special notices . . . . .</b>	<b>77</b>
<b>Appendix D.</b>	<b>Related publications . . . . .</b>	<b>79</b>
D.1	IBM Redbooks . . . . .	79
D.2	IBM Redbooks collections . . . . .	79
D.3	Other resources . . . . .	79
<b>How to get IBM Redbooks . . . . .</b>	<b>81</b>	
IBM Redbooks fax order form . . . . .	82	
<b>Glossary . . . . .</b>	<b>83</b>	
<b>Index . . . . .</b>	<b>85</b>	
<b>IBM Redbooks review. . . . .</b>	<b>89</b>	

---

## Preface

This IBM Redbook describes XPLink, the new OS/390 high performance linkage option. It discusses the means by which XPLink achieves its performance goals, and the various ways these affect the performance of C and C++ code. Finally, it shows the effect of the XPLink performance improvements on real-world applications.

---

### The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization Poughkeepsie Center.

**Michael MacIsaac** is a team leader for S/390 redbooks and workshops at the ITSO Poughkeepsie Center. He writes about and teaches classes on OS/390 UNIX and Linux for S/390. Michael has worked at IBM for 13 years, mainly as a UNIX programmer.

**Graham Ewart** is a Senior Software Developer at the IBM Toronto Laboratory in Canada and a member of the team that designed XPLink. He has 35 years of experience in computing and data processing, mostly in software development. Graham has worked at IBM for 24 years. He holds a degree in Mathematics and Physics from Queen's University at Kingston, in Kingston, Ontario.

**Art Goldschmidt** is a Senior Programmer at the IBM Poughkeepsie Development Laboratory in the USA. He has 36 years of experience with IBM in operating systems, chip and mechanical design systems, Internet supply chain management, open systems standards, and performance tools and techniques. He is the author of several dozen publications and holds a MSEE from Syracuse University.

**Dave Sudlik** is an Advisory Software Engineer at the IBM Poughkeepsie Development Laboratory in the USA. He has 14 years of experience with IBM, with the last 5 spent in Language Environment development. He holds an MS degree in Computer Science from Rensselaer Polytechnic Institute.

Thanks to the many others who contributed to this project.

Special thanks are due to two people who contributed input both in terms of test case results and meticulous reviews:

- David Cargill, IBM Toronto
- Jean-Louis Lafitte, IBM France

Thanks to the following people who were involved in the architecture, design, and implementation of XPLink:

- Leona Baumgart, IBM US
- Hans Böttiger, IBM Germany
- Pat Healey, IBM US
- Martin Hopkins, IBM Research
- Mike Ludwig
- Jim Mulvey, IBM US
- Greg Reid, IBM Canada

Thanks to the following people who provided measurement inputs to this redbook:

- Joe Bostian, IBM US
- Maury Clark, IBM US
- Raymond Mak, IBM Canada
- Kevin McKenzie, IBM US
- Sonomi Mukaida, IBM Japan
- Rich Prewitt, Jr., IBM US
- Daniel Prevost, Data Kinetics, Ltd.

Thanks to Rich Conway, Cindy Curley, Bob Haimowitz, and Dotti Still  
International Technical Support Organization, Poughkeepsie Center

Thanks also to Terry Barthel, Alison Chandler and Alfred Schwab for their  
editorial assistance  
International Technical Support Organization, Poughkeepsie Center

---

## Comments welcome

### **Your comments are important to us!**

We want our Redbooks to be as helpful as possible. Please send us your comments about this or other Redbooks in one of the following ways:

- Fax the evaluation form found in “IBM Redbooks review” on page 89 to the fax number shown on the form.
- Use the online evaluation form found at [ibm.com/redbooks](http://ibm.com/redbooks)
- Send your comments in an Internet note to [redbook@us.ibm.com](mailto:redbook@us.ibm.com)



---

## Chapter 1. Overview

XPLink was designed to bring performance improvement opportunities to a large class of applications: call-intensive C and C++ applications typical of the newer workloads being seen on OS/390.

XPLink achieves this by changing register conventions and the layout of the stack. These changes allow for:

1. Faster detection of stack overflow
2. Faster saving of registers
3. Faster allocation of local storage for called functions
4. Opportunities for improved register allocation in function bodies
5. Greater use of registers for function arguments and return values, improving the performance of argument list construction, parameter use, return value construction, and return value use

Both the measurements of small code fragments in Chapter 5, "Performance characteristics" on page 39 and the measurements of real applications in Chapter 6, "Application and benchmark measurements" on page 53 show that XPLink has met its design goals. Some of the highlights of this are:

- Measurements of many specific code fragments changed by XPLink show *over 30% performance improvement* for the specific fragments measured (see, for example, 5.2.1, "The effect of parameter-list size" on page 42).
- In mixed COBOL-C applications there are performance opportunities that may rely heavily on application restructuring. In a synthetic test having a very high proportion of very small C functions, the measurements seemed to converge at an *improvement of well over 40%* (see 5.4, "The cost of glue code" on page 45).
- The Lotus Domino server requires *over 20%* more CPU power running without XPLink (see 6.3.1.2, "Mail and Calendaring Benchmark - 17.8% CPU savings" on page 64).
- An independent TCP/IP-based table management product shows a *2-20 % improvement* (see 6.4.2, "net.TABLES from Data Kinetics Ltd." on page 66).

The first two measurements are entirely synthetic and not likely to be seen in real applications; the second pair are *real measurements of real applications*.

However, there remains a large body of application code running on OS/390 that will not benefit from XPLink and that should not be converted to XPLink without very careful consideration of the costs and risks involved. This includes:

- Applications with large, complex assembler-language components
- Other mixed-language applications
- Applications with very large functions or little function calling
- Many applications that make high use of the run-time library

Over time, we can expect to see the difficulties associated with using Assembler Language in an XPLink environment diminish. Furthermore, the disadvantages of using the C/C++ run-time library will diminish in future releases as an increasing proportion of the run-time is implemented in XPLink; already we have the IEEE floating point mathematical functions converted to XPLink so they can be invoked without stack swapping (see 6.2.2.3, "Math library interfaces: IEEE binary float

and hex float” on page 58). We expect most new run-time function to be implemented and delivered exclusively in XPLink.

XPLink is one of many initiatives aimed at improving the performance of C and C++ applications on OS/390. Others include:

- Exploitation of improvements in S/390 processors.
- Continued focus on improving optimization techniques based on compiler research, customer feedback, and examination of code sequences emitted by the compiler.
- Interprocedural Analysis (IPA), a compiler option which, among other things, finds ways to reduce function call overhead by reorganizing code across an entire application, eliminating function calls where possible.

To a large extent, the results of these initiatives are additive, the exception being XPLink and IPA, which take separate, independent approaches to reducing function call overhead.

In many applications, IPA may provide greater benefits than XPLink. IPA, however, is generally the most expensive of these in terms of compile-time resources and build-time investment, which may or may not pay off. XPLink and IPA cannot be combined in C/C++ for OS/390 Version 2 Release 10, however we expect this restriction to be removed in a future release.

In the meantime, XPLink offers a real performance benefit of 20% or more to many C/C++ applications, at little or no cost in application build time.

---

## Chapter 2. Technical overview

XPLink is a new linkage convention for OS/390. It has many benefits, but one purpose: performance. It is not intended to replace the traditional linkage conventions in today's OS/390 environment; rather it is intended for use in applications where performance considerations outweigh all other needs.

Applications can contain a mixture of code compiled with XPLink and code compiled with today's OS/390 linkage conventions, but there are restrictions in where this can be done and run-time performance costs associated with doing this.

The purpose of this book is to illustrate the benefits of using XPLink and to guide the reader in evaluating the benefits of XPLink in various contexts.

---

### 2.1 History

The current calling conventions for High Level Languages (HLLs) on OS/390 are an outgrowth of the original OS/360 Assembler Language conventions first introduced in 1964. These conventions specify that, on entry to a subroutine:

- Register 13 points to an 18-word save area aligned on a full word boundary
- Register 14 points to a return point in the calling routine
- Register 15 contains the entry point address of the routine being called
- Register 1 points to a list of addresses, each being the address of an actual argument (specified in order) and the last, only, having its high-order bit on

Called subroutines are responsible for saving and restoring any General Purpose Registers (GPRs) they might alter in well-defined locations in the save area provided by the caller. Furthermore, a called routine that makes further calls must acquire a save area of its own to provide to any routine it might call. In this case, the called function is responsible for linking the two save areas together in a doubly-linked list: the address of the caller-provided save area being stored at offset 4 in the newly-acquired save area (this is called the back chain) and the address of the newly-acquired save area being stored at offset 8 in the caller-provided save area (the forward chain).

These conventions are described in more detail in *OS/390 V2R9.0 MVS Assembler Services Guide*, GC28-1762, found at:

<http://ibm.com/s390/bookmgr-cgi/bookmgr.exe/BOOKS/IEA1A620/2.0>

In an environment that requires recursion or reentrancy, this convention requires that routines dynamically acquire (via GETMAIN) save areas and storage for local variables at some cost in performance. In the late 1960s HLLs began to overcome this cost by acquiring a large block of storage (the Stack Segment or Initial Stack Allocation (ISA)) at initialization and carving out space for save areas and local variables (together called the Dynamic Storage Area (DSA) or stack frame) from this block at the beginning of each routine.

This mechanism was first introduced by PL/I, carried forward to the C/PLI common run-time library with the introduction of C/370 in 1988, and from there into Language Environment for MVS and VM (LE). This mechanism is described

in *OS/390 V2R10.0 Language Environment for OS/390 & VM Vendor Interfaces*, SY28-1152.

In the LE environment, on entry to a routine, GPR 13 points to an area (the DSA) that begins with a standard 18-word save area. This save area is followed immediately by control information and the local (automatic) storage belonging to the calling function. The control information includes the address of the first byte following this DSA. The called function need only load this address from a fixed offset from GPR 13, verify that there is enough space there for its own DSA, and use that as its own save area (and DSA) address. The layout of an LE stack frame is shown in Figure 1.

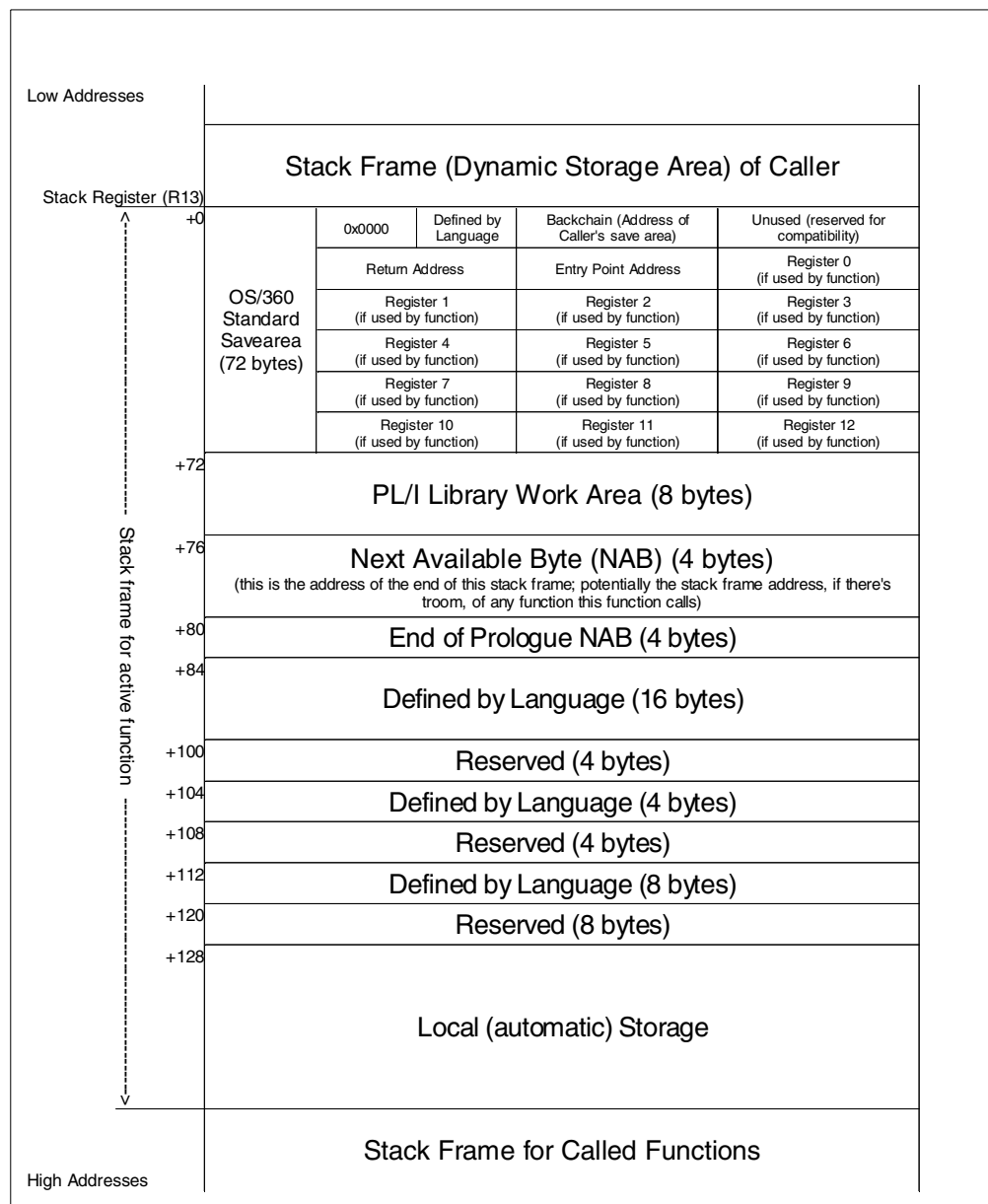


Figure 1. Language Environment stack frame layout

Note that updating the stack frame address is shared between the caller and the called function: the caller computes the address of the stack frame for called

functions and the called functions load this computed address and verify that their stack frames fit in the space available.

This is called an “upward-growing stack;” every routine’s DSA, if it fits in the Stack Segment, is at a higher address than its caller’s DSA.

This provides an upward-compatible linkage in the sense that LE-conforming routines pass the address of an 18-word area in GPR 13, allowing such routines to call assembler code written to the original OS/360 conventions. It does not allow such assembler code to call LE-conforming HLL code: the control information that contains the end of the current DSA is missing from the OS/360 conventions. Of course assembler code can, with change, masquerade as LE-conforming HLL code; LE provides assembler macros for this specific purpose.

---

## 2.2 What is new

XPLink introduces several changes to the conventions used to date by LE-conforming HLLs:

- The stack grows towards lower addresses (a downward-growing stack).
- The mechanism for determining if a called routine’s stack frame will fit in the current stack segment has changed.
- The register conventions have changed.
- The argument-passing conventions have changed.

These changes will be discussed in detail later.

XPLink is supported by the OS/390 C/C++ compiler, and to a limited extent by the High Level Assembler. There is no support announced by other LE-conforming HLLs.

---

## 2.3 Performance benefits of the new calling conventions

XPLink is a complete overhaul of OS/390 calling conventions, incorporating many changes, each designed to increase the performance of applications.

### 2.3.1 Stack organization

The first change is the organization of the stack. There are several significant changes relating to the stack:

1. The stack grows towards lower addresses.
2. Information needed to move from one stack frame to another is stored in static data, not produced dynamically during run-time.
3. The mechanism for determining that the stack is out of room has changed.
4. Register conventions have changed.
5. Register saving conventions have changed.

#### 2.3.1.1 The downward-growing stack

In the traditional upward-growing stack environment, stack frames for functions on the call stack occupy contiguous areas<sup>1</sup> of storage in the LE-managed stack segment. The difference between the address of a caller’s stack frame and the address of a called function’s stack frame is the size of the caller’s stack frame.

<sup>1</sup> Ignoring, for now, the possibility that stack frames may not all fit into the same stack segment.

The responsibility for updating the pointer to the current stack frame (GPR13 in the case of current OS/390 conventions) rests with the called function but, in general, the called function does not know which function is calling it or how big the caller's stack frame is. This requires that callers contain instructions to compute and store the address of a stack frame to be used by functions they might call, and that every called function contain instructions to find this value and verify that there is enough room in the stack segment for its own stack frame.

With a downward-growing stack, as used in XPLink, the amount by which a called function must update the stack pointer is the size of its own stack frame, a value known at compile time. There is no need for stack sizes or addresses to be passed around in storage for a function call; the called function can compute the new stack address by subtracting its own stack frame size from the current stack pointer.

This change saves the cost, incurred on *every* function call, of computing, storing, and loading stack frame sizes or addresses.

### 2.3.1.2 Static stack information

In current OS/390 conventions, each save area (stack frame) contains the address of the previous stack frame, stored at offset 4. Every function's prolog contains instructions to store this value, the backchain, in its own stack frame. This allows tools, and dump readers, to follow the call stack to determine, for example, all currently active functions ("who called whom").

In XPLink, this information (the backchain) is not normally stored. Instead, tools and dump readers must find the size of the stack frame for each currently active function (stored in read-only control information stored with the function) by following pointers to the code. The Interactive Problem Control System (IPCS), described in *OS/390 MVS Interactive Problem Control System (IPCS) User's Guide*, GC28-1756 at:

<http://ibm.com/s390/bookmgr-cgi/bookmgr.exe/BOOKS/IEA1C610/CONTENTS>

provides a framework for following stack frames. This is exploited by the Language Environment LEDATA IPCS Verbexit described in *OS/390 Language Environment for OS/390 & VM Debugging Guide and Run-Time Messages* (SC28-1942-09 or later) at:

<http://ibm.com/s390/bookmgr-cgi/bookmgr.exe/BOOKS/CEEA1030/CCONTENTS>

and more specifically at:

<http://ibm.com/s390/bookmgr-cgi/bookmgr.exe/BOOKS/CEEA1030/1.3.5>

The underlying data structures used here are described in *OS/390 Language Environment for OS/390 & VM Vendor Interfaces* (SY28-1152-08 or later) at:

<http://ibm.com/s390/bookmgr-cgi/bookmgr.exe/BOOKS/CEEV1000/2.4.3.5>

XPLink provides a compiler option (`XPLINK(BACKCHAIN)`) to generate instructions in every function's prolog that cause a backchain field to be stored in every stack frame; the performance impact of this option is discussed in "The effect of XPLink suboptions on performance" on page 44. This option is provided for the peace of mind of individuals faced with a paper storage dump and no tool.

### 2.3.1.3 Stack overflow detection

Current LE stack conventions require that compilers generate instructions in every function's prolog to check that there is sufficient room in the current stack segment for that function's stack frame. XPLink avoids this by allocating a protected 4K page-aligned range of virtual addresses immediately prior to the stack segment. This range of virtual addresses is called the Guard Page. XPLink code detects stack overflow by attempting to store into the beginning of every function's stack frame in the function prolog; if the stack segment is not large enough this attempt will cause a hardware interrupt that will cause LE to allocate another stack segment for this function, hook everything up to make it appear that nothing untoward had happened, and continue.

This method works only if the current function's stack frame is smaller than the guard page (4K); the compiler uses other strategies if this is not the case. The performance implications of these strategies are discussed in 5.2, "The effect of stack size on XPLink performance" on page 40.

Considerable savings are realized by not making specific comparisons for stack overflow, but the cost of running out of space in a stack segment is considerable. ***It is impossible to overemphasize the importance of tuning the initial stack allocation (specified by the `STACK` run-time option) of an application to avoid this cost.***

Two options are provided to tune the stack requirements of XPLink applications:

1. A compiler option (`XPLINK(NOGUARD)`) which causes the compiler to generate instructions in every function's prolog that specifically compare the function's beginning-of-stack address with the stored stack-segment- beginning address
2. A run-time option (`RPTSTG(ON)`) which causes a storage report showing, among other things, the actual stack utilization of the running application

These options should be used together during application deployment to determine the optimal initial stack allocation. Depending on the size of a function's stack frame, the `NOGUARD` suboption can add to the cost of a function call and thus to the overall time taken to run the application. The effect of this, in a trivial "application," can be seen in Figure 4 on page 42 and in the following figures. Except for applications having unpredictable stack requirements, this option should be removed for production when tuning is complete. See 3.6, "Storage tuning" on page 21.

It is not unreasonable to include the initial stack allocation in the application's code (via `#pragma runopts(stack(...))`) to avoid having to specify the correct stack allocation on every invocation of the application.

In XPLink, the overhead of the `RPTSTG(ON)` run-time option is substantially lower than in a non-XPLink environment. Leaving this run-time option in production code to help diagnose performance problems in applications that are not properly tuned should be considered.

### 2.3.1.4 New register conventions

XPLink has new register conventions: instead of GPR 13 pointing to the current stack frame (or caller's stack frame on entry), GPR 4 is used. There are other register conventions: GPR 5 points to the called function's environment (its own part of the Writeable Static Area); GPR 6 is used, where required, as the entry point of the called function; and GPR 7 is the return address. Furthermore, GPR

4, instead of pointing directly to the stack frame, points to 2K bytes before the stack frame. This allows function prologs to store into their own stack frame (this is the instruction that checks for stack overflow) prior to updating GPR 4, providing for a faster-executing instruction stream.

### 2.3.1.5 Register saving conventions

In today's linkage conventions the GPRs used by a function are saved in a save area provided by the caller. In XPLink, these registers are saved in the called function's own stack frame, the first 12 words being reserved for saving GPRs 4 to 15 as required.

As with current conventions, the XPLink conventions do not set aside areas to store floating point or access registers. These registers, if used, are saved in areas carved out of the called function's own automatic storage.

XPLink and non-XPLink register conventions are shown in Table 1.

Table 1. Register conventions at function entry

General Purpose Register	XPLink conventions	LE conventions
0	Not preserved over a function call	Writeable Static Area (WSA)
1	1st argument, not preserved over a function call	Address of argument list
2	2nd argument, not preserved over a function call	Work registers, preserved over a function call
3	3rd argument, not preserved over a function call	
4	Stack register; points to 2K before the active stack frame	
5	Points to the function's "environment", its "@STATIC" part of the Writeable Static Area	
6	Work register, often used to point to the called function's entry point, not preserved over a call	
7	Return address, not preserved over a function call	
8-11	Work registers, preserved over a function call	
12	LE Common Anchor Area (CAA), preserved over a function call	
13	Work registers, preserved over a function call	Stack register
14		Return address, preserved over a function call
15		Called functions entry point

## 2.3.2 Stack layout

The layout of the XPLink stack is shown in Figure 2.



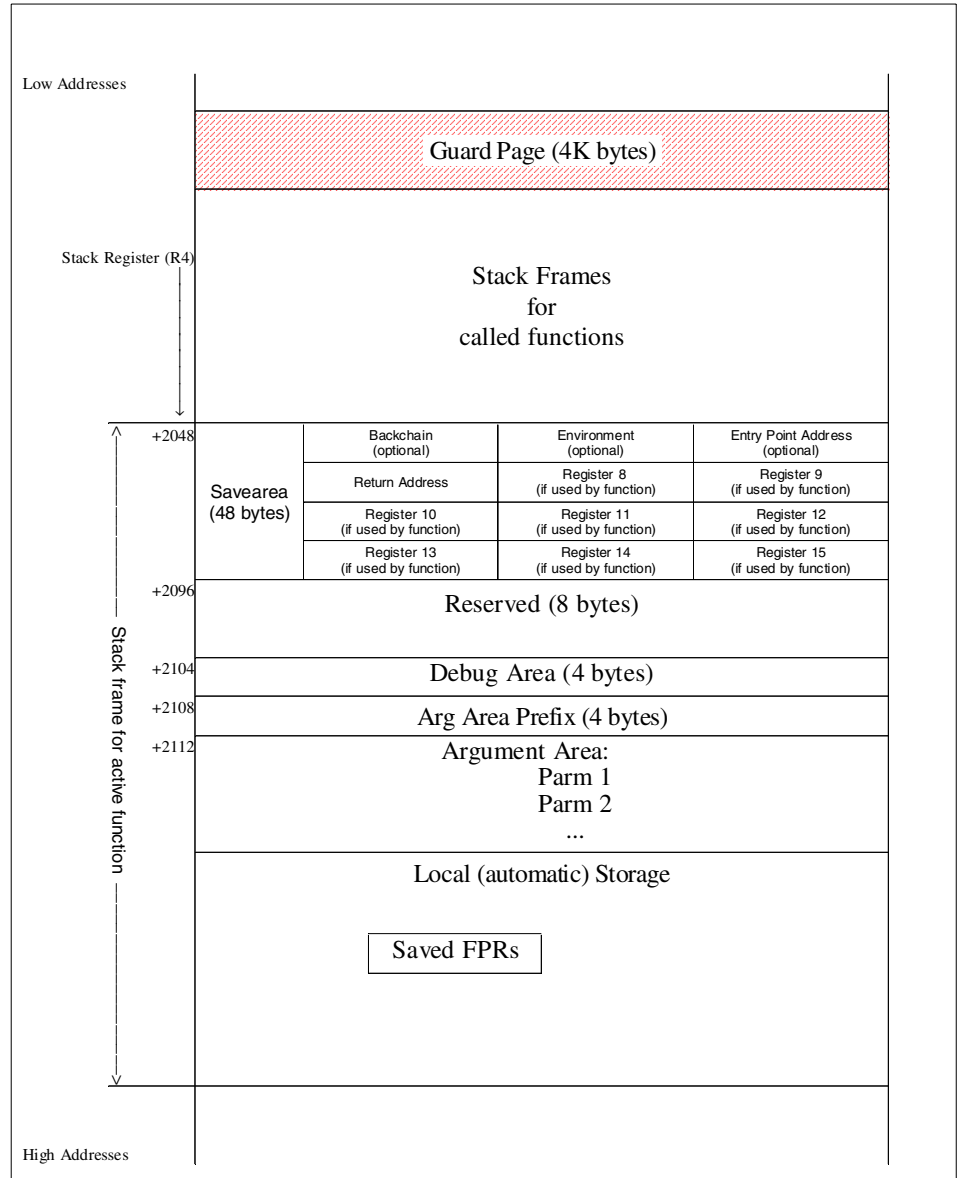


Figure 2. The XPLink stack frame

### 2.3.3 Argument passing conventions

The second significant change introduced with XPLink is its argument-passing conventions. The two significant changes are:

1. The argument list is constructed in a fixed location, called the Argument Area, in the caller's stack frame.
2. Some arguments are passed in registers instead of in the argument area.

#### 2.3.3.1 The Argument Area

Current conventions specify that on entry to a function GPR 1 contain the address of the function's parameter list in storage. This may be a by-reference or by-value list. In the former case GPR 1 points to a list of 4-byte addresses, the first being the address of the first actual argument, the second being the address

of the second, and so on up to the last address which, in addition to pointing to the last actual argument, has its high-order bit turned on.

In the latter case, GPR 1 points to a list of actual argument values arranged in contiguous locations in storage. Some of these may be addresses, of course; the application is free to pass pointers as arguments or, in the case of C++, reference parameters indicated in the function prototype by an ampersand (&). There is no mechanism for indicating the last argument with by-value parameter passing.

In XPLink, instead of requiring that GPR1 point to a parameter list, the parameter list is in a fixed location (offset +64) in the caller's stack frame. This allows functions to treat their incoming parameters (actual arguments) as an extension of their own local (automatic) storage, using the same register to address both. This makes an additional register available to the compiler for optimization in the function's body.

### **2.3.3.2 Arguments passed in registers**

For performance reasons, up to 4 floating point arguments are passed in Floating Point Registers (FPRs) 0, 2, 4, and 6. The first 3 words (12 bytes) of the argument area are passed in GPRs 1-3. Arguments passed in registers are not passed in the argument area, although space is reserved there for them.

Except that the registers containing incoming actual arguments are pre-assigned by the XPLink linkage conventions, the behavior of these arguments is identical to any other local (automatic) data item that may have been put into a register by the optimizer: they may be stored in their home location (in this case, the caller's argument area) if the compiler determines that there is a better use for the register currently holding their values, or their values may be discarded without being saved if there is no further use for them.

Sometimes debugger users or dump readers like to be able to determine the values of the actual arguments to a function. The compiler provides an option, `XPLINK(STOREARGS)` which causes it to generate instructions in every function's prolog that specifically store incoming register arguments in their home locations. The performance implication of this suboption on function prologs is discussed in 5.3, "The effect of XPLink suboptions on performance" on page 44.

---

## **2.4 Future of non-XPLink**

There is a large body of code written and compiled using today's linkage conventions. This code represents a valuable asset to its owners; it's not going to go away.

Furthermore, much of this code is written in languages that do not support XPLink.

In the environments where this code runs, the OS/390 C/C++ compiler will continue to support today's linkage conventions. In fact, this is the default behavior. Furthermore, LE and the C/C++ compiler will support run-time compatibility between XPLink and non-XPLink code in limited environments.

XPLink cannot be combined with Inter-Procedural Analysis in C/C++ for OS/390 Version 2 Release 10. We expect this restriction to be removed in a future release. However, you should not expect the effects of XPLink and IPA to be

additive: IPA removes opportunities for function call improvement from an application by, among other things, making inline copies of functions at their call sites.

However, new features added to the C/C++ Compiler or to LE in the future may be supported in an XPLink environment only.



---

## Chapter 3. Terminology and usage

This chapter discusses issues with installing and customizing the XPLink environment. It also describes the steps taken to build, run, debug, and tune an XPLink application.

Some aspects of an application might make it unsuitable for conversion to XPLink. These are discussed in detail in Chapter 4, “Compatibility considerations” on page 23.

---

### 3.1 Installation and customization

Beginning in OS/390 V2R10, there are a number of new library data sets that ship as a part of Language Environment (LE). This discussion summarizes the roles of both existing and new data sets, and discusses their relevance to XPLink.

The first category of data sets discussed here contain what is referred to as *resident routines*. Resident routines are statically linked with the application during the link-edit step, and contain such things as initialization/termination routines and pointers to callable services.

- SCEELKED** This library contains LE resident routines for non-XPLink applications, including those for callable services, initialization, and termination. This includes language-specific callable services, such as those for the C/C++ run-time library. Only case-insensitive names of eight or fewer characters in length are contained in this library. *This library must be used only when link-editing a non-XPLink program.*
- SCEELKEX** Like SCEELKED, this library contains LE resident routines for non-XPLink applications. However, case-sensitive names which can be greater than eight characters in length are contained in this library. This allows symbols such as the C/C++ `printf()` and `pthread_create()` functions to be resolved without requiring the names to be converted to upper case, truncated, or mapped to another symbol. *This library must be used only when link-editing a non-XPLink program.*
- SCEE OBJ** This library contains LE resident definitions for non-XPLink applications which may be required for OS/390 UNIX System Services (OS/390 UNIX) programs. *This library must be used only when link-editing a non-XPLink program.*
- SCEECPP** This library contains LE resident definitions for non-XPLink applications which may be required for C++ programs, such as the definition of the `new` operator. *This library must be used only when link-editing a non-XPLink program which includes any NOXPLINK-compiled C++ object modules.*
- SCEEBIND** This library contains all LE resident routines for XPLink applications. This one library replaces the four libraries of resident routines for non-XPLink applications. When link-editing XPLink applications, this one library is used wherever one of the non-XPLink resident routine libraries (SCEELKED, SCEELKEX, SCEE OBJ, or SCEECPP) had been used. It provides only a

small number of resident routines, since most of the functions formerly provided in those static libraries are instead provided using dynamic linkage. *SCEEBIND must be used only when link-editing an XPLink program.*

**SCEELIB** This library contains side-decks for Dynamic Link Libraries (DLLs) provided by LE. Many of the language-specific callable services available to XPLink applications appear externally as DLL functions. (Refer to *OS/390 Language Environment Programming Guide*, SC28-1939 for information on using DLLs.) To resolve references to these callable services from an XPLink application, a definition side-deck must be included when link-editing the application. The SCEELIB library contains the following side-decks:

- **CELHS001:** Side-deck to resolve references to LE services when link-editing an XPLink application. This includes both Application Writer Interfaces (AWIs, see *OS/390 Language Environment Programming Reference*) and Compiler Writer Interfaces (CWIs, see *OS/390 Language Environment Vendor Interfaces*). The entries in this side-deck replace the corresponding non-XPLink resident routines in SCEELKED. The AWI stubs also exist as executables in SCEERUN, which can be loaded and run from non-XPLink applications. This technique cannot be used with XPLink applications.
- **CELHS003:** Side-deck to resolve references to callable services in the C/C++ run-time library when link-editing an XPLink application. The entries in this side-deck replace the corresponding non-XPLink resident routines in SCEELKEX, SCEELKED and SCEEOBJ.
- **CELHSCPP:** Side-deck to resolve references to C++ run-time library (RTL) definitions that may be required when link-editing an XPLink application. The entries in this side-deck replace the non-XPLink resident routines in SCEECPP.

The second category of data sets discussed here contain what is referred to as *dynamic routines*. Dynamic routines are not part of the application and are loaded dynamically during run-time.

**SCEERUN** This is a PDS which contains the run-time library routines needed during execution of applications written in C/C++, PL/I, COBOL and FORTRAN.

**SCEERUN2** This is a PDSE which contains the run-time library routines needed during execution of applications written in C or C++.

Note that the SCEERUN2 data set is a PDSE. OS/390 V2R10 marks the first OS/390 release that ships program products in PDSE data sets.

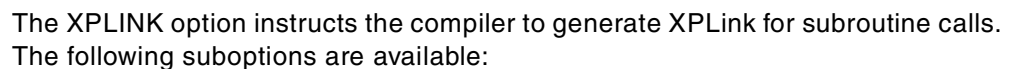
By definition, both the SCEERUN and SCEERUN2 data sets are part of the LE dynamic library, and should be available at application run-time, either from LNKLIST or STEPLIB (Run-Time Library Services (RTLS) is not supported under XPLink.) However, in OS/390 V2R10 the SCEERUN2 data set consists of dynamic routines that are required solely to run XPLink applications. But future

Applications that currently use the STEPLIB environment variable for the SCEERUN data set to gain access to the run-time library provided by LE do not need to add the SCEERUN2 data set as part of their STEPLIB concatenation. In fact, since SCEERUN2 contains module names that do not intersect with any pre-LE run-time library or any existing library, IBM recommends that SCEERUN2 be added to the LNKLIST. This will not result in any adverse effects.

Refer to *OS/390 V2R10.0 Language Environment for OS/390 Customization*, SC28-1941 for more details and sample procedures.

This section describes the steps required to compile and bind an application for XPLink.

XPLink applications can be created by using the OS/390 C/C++ Compiler. There is a new XPLINK, compiler option (shown here with the minimal abbreviation):



Chapter 3. Terminology and usage 15

NOBACKCHAIN is the IBM-supplied default.

**STOREARGS | NOSTOREARGS:** used to specify that the compiler insert extra code to explicitly save parameters that are passed in registers into their natural locations in the incoming argument area. This tends to preserve the actual incoming argument values longer; perhaps, but not always, for the entire life of the called function. There are no IBM-supplied products that require or take advantage of the fact that STOREARGS has been specified. It is up to the person doing the debugging to know that STOREARGS has been turned on so that the parameters can possibly be more easily located in a dump by examining the caller's argument area.

NOSTOREARGS is the IBM-supplied default.

**GUARD | NOGUARD:** used to specify that the compiler insert extra code to check for end-of-stack by using control information stored in LE control blocks rather than by attempting to store into the beginning of the new stack frame.

GUARD is the IBM-supplied default.

**OSCALL(Upstack | Downstack | Nostack):** used to specify the meaning of `extern "OS"` (for C++) or `#pragma linkage(...,OS)` for C. For more details refer to *OS/390 C/C++ User's Guide*, SC09-2361-06 or later.

Note that use of BACKCHAIN, STOREARGS, and NOGUARD each causes the generated code to be slower. The IBM-supplied defaults create the fastest code.

Unless explicitly stated, none of these suboptions was specified (that is, the defaults applied) for the runs that produced the performance information in this book.

When compiling using the XPLINK option, the compiler uses the following options as defaults:

- CSECT()
- GOFF
- LONGNAME
- RENT

You may override these options. However, the XPLINK option requires the GOFF option. If you specify the NOGOFF option with XPLINK, the compiler issues an informational message and overrides the option to GOFF.

In addition, the XPLINK option requires that the value of ARCH must be 2 or greater. The compiler issues an error message if you specify `ARCH(0)` or `ARCH(1)` with XPLINK.

XPLink applications are automatically enabled for DLLs. There is no difference between XPLink code that calls a DLL and code that does not. Therefore, you never need to specify the DLL compile option when you specify the XPLINK compile option.

### 3.2.2 Binder support

XPLINK-compiled objects must be link-edited using the DFSMS Version 1 Release 6 Program Management Binder. There is no support in the prelinker for XPLINK-compiled objects.



The binder output of an XPLink application is a program object in PM3 format, and must reside in either a PDSE or the Hierarchical File System (HFS). An XPLink application cannot reside in a PDS.

There are no new binder options for XPLink.

Calls between functions compiled XPLink and non-XPLINK are supported, for example, using the DLL call mechanism. But when binding together objects to form an executable, the objects normally must all be either XPLINK-compiled or NOXPLINK-compiled (see 4.1, “Mixing XPLink and non-XPLink code” on page 23). You cannot mix XPLink and non-XPLink objects in the same program. The binder enforces this by ensuring the linkage attributes of a reference to a function match the linkage attributes of the definition of that function. If they do not match, the binder issues the message:

```
IEW2469E THE ATTRIBUTES OF A REFERENCE TO symbol-name DO NOT MATCH THE
ATTRIBUTES OF THE TARGET SYMBOL.
```

Typically, this message occurs when a non-XPLink compilation unit calls a function in a compilation unit that was compiled XPLink (or vice versa), and these compilation units are trying to be statically linked. Make sure all source parts have been compiled XPLink if you’re trying to create an XPLink executable.

Another way this message could be received is when a C or C++ function that has been recompiled XPLink attempts to call an assembler function that has not been converted to XPLink. You must either convert the assembler function to XPLink (see 4.2.1, “XPLink assembler” on page 24), or indicate that this call should use non-XPLink OS Linkage calling conventions (see 4.2.2, “Non-XPLink assembler” on page 28).

This message would also be issued if an XPLink application is mistakenly bound using the SCEELKED data set. The LE resident routines for non-XPLink applications are assembler “stub” routines with non-XPLink entry points. To avoid this problem, make sure you invoke `c89` or `c++` commands with the `-Wl,xplink` link-edit phase option. This tells these commands to use the SCEEBIND and SCEELIB data sets.

Use the binder options `LIST=NOIMP`, `MAP,XREF` and examine the binder map and xref output to determine the source parts containing the mismatched function reference and definition.

For more information on the binder, refer to *OS/390 DFSMS Program Management*, SC27-0806.

### 3.2.3 Building XPLink applications from a UNIX shell

From the shell, the `c89` and `c++` commands are used to compile C and C++ programs. XPLink applications can be created by using the `-W` option to specify additional options to the compile and link-edit phases. For example, to compile a simple C program as an XPLink application, you could use:

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

Specifying XPLINK on the compile phase (`-Wc`) indicates to the compiler to use the XPLINK compile option. You can also specify any suboptions that you require at this time.

Specifying XPLINK on the link-edit phase (-Wl) does not pass an XPLINK option to the binder. There are no new binder options for XPLink. Instead, specifying XPLINK on the link-edit phase tells c89/c++ to use the LE SCEEBIND data set containing XPLink resident routines instead of the normal non-XPLink (SCEELKED, SCEELKEX, SCEEOBJ, and SCEECPP) data sets. Refer to the `c89` command description in the *OS/390 V2R10.0 UNIX System Services Command Reference*, SC28-1892.

Note that even though an XPLINK-compiled program is implicitly a DLL-compiled program, you must still specify DLL on the link-edit phase to produce a side deck and make the executable usable as a DLL. For example:

```
c89 -o MyDll -Wc,xplink -Wl,xplink,dll MyDll.c
```

### 3.2.4 Building XPLink applications with JCL

XPLink applications are compiled using one of the following IBM-supplied catalogued procedures: CBCC, CBCXCB, CBCXCBG for C++; EDCC, EDCXCB, EDCXCBG for C; or equivalent. When CBCC or EDCC is used, or when an XPLINK suboption is required, the XPLINK compiler option must be specified using the CPARM symbolic parameter.

XPLink applications are bound using one of the following IBM-supplied catalogued procedures: CBCXCB, CBCXCBG, CBCXB, CBCXBG for C++; CEEXL, CEEXLR, EDCCXB, EDCCXBG for C; or equivalent.

These catalogued procedures can be found in the data sets CBC.SCBCPRC and CEE.SCEEPROC.

### 3.2.5 Building XPLink applications under TSO

XPLink applications are compiled using one of the following IBM-supplied REXX utilities: CXX for C++; CC for C; or equivalent. The XPLINK compiler option must be specified as an option.

XPLink applications are bound using the CXXBIND IBM-supplied REXX utilities. Specify the XPLINK option to get the LE XPLink data sets instead of the default non-XPLink data sets.

These REXX utilities can be found in the data set CBC.SCBCUTL.

---

## 3.3 Running an XPLink application

For the most part, there is nothing special that has to be done to run an XPLink application.

The following data sets are required at run-time, and will be searched for (in order) in the current STEPLIB/JOBLIB, LPA, or LNKLIST:

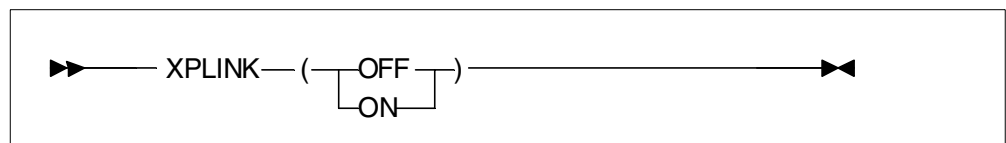
1. The data set containing the application code
2. CEE.SCEERUN
3. CEE.SCEERUN2
4. CBC.SCLBDLL (for C++ applications that use the IBM Class Library DLLs)

These data sets are in the STEPLIB concatenation when you use the IBM-supplied CBCXG catalogued procedure. Furthermore, if the main application

is *not* XPLink and the application uses a DLL that *is* XPLink, the LE XPLINK(ON) run-time option must be specified. This can be specified using the GPARM symbolic parameter in the IBM-supplied catalogued procedures.

When an XPLink application is run, LE needs to know to allocate the XPLink run-time environment. Normally, the initial program can be inspected for its linkage attributes, and if it's XPLink, then a downward-growing stack is established and the XPLink C/C++ Run-Time Library (RTL) is loaded.

There is one case where this is not possible, and that is when a non-XPLink initial program (eg. `main()`) will at some point during its execution call an XPLink function in another program object. This could be done by calling a function in a DLL. In this case, LE needs to know at initialization time that XPLink resources must be acquired for use later on in the execution of the application. That is the purpose of the XPLINK run-time option.



If your initial program is an XPLink application, then the XPLINK run-time option will be set ON. No message will be issued to indicate this action. In this case a run-time options report from RPTOPTS will indicate OVERRIDE on the XPLINK option.

If your initial program is a non-XPLink application and you don't specify the XPLINK(ON) run-time option, then no XPLink resources will be allocated at initialization time. If your application attempts to call an XPLink function, maybe through a call to a DLL, your application will be terminated with the message:

```
CEE3555S A call was made from a NOXPLINK-compiled application to an
XPLINK-compiled exported function in DLL <dllname> and the XPLINK(ON)
runtime option was not specified.
```

By specifying XPLINK(ON) for this application, an XPLink environment will be established at initialization time and the subsequent call to an XPLink DLL function will succeed.

It's very likely that the performance of predominantly non-XPLink applications will be degraded if run with the XPLINK(ON) run-time option. It's also possible that a non-XPLink application won't run at all if it uses resources or subsystems that are restricted in an XPLink environment (for example, AMODE 24). For these reasons, the XPLINK(ON) run-time option should only be specified when necessary, on an application-specific basis. In fact, it is not possible to specify XPLINK(ON) as an installation default (in the CEEDOPT) or in the assembler user exit (CEEBXITA). You can specify XPLINK(ON) in a CEEUOPT, with a `#pragma runopts` directive, or as an option at program invocation. You can also specify XPLINK(ON) through the `_CEE_RUNOPTS` environment variable, but it is highly recommended that you do not export `_CEE_RUNOPTS` globally in this case.

When an application is running in an XPLink environment (that is, either the initial program was XPLink or the XPLINK(ON) run-time option was specified), the

ALL31 run-time option will be forced to ON and the STACK run-time option will be forced to STACK(,ANY). No AMODE 24 routines are allowed in an enclave that uses XPLink, and the stack can be obtained anywhere in storage. No message will be issued to indicate these actions. In this case a run-time options report from RPTOPTS will indicate OVERRIDE on the ALL31 and STACK options.

For more information on LE run-time options, refer to the *OS/390 V2R10.0 Language Environment for OS/390 and VM Programming Reference*, SC28-1940.

---

### 3.4 Debugging an XPLink application

Debugging techniques for XPLink applications are very similar to those used for non-XPLink applications. There is now the added responsibility of having to recognize the linkage conventions of the function being debugged. LE Tracebacks, as provided with CEEDUMP and the IPCS LEDATA VERBX command, clearly identify each stack frame as UPSTACK (non-XPLink), DOWNSTACK (XPLink), or TRANSITIONAL (use when switching between stack types, for stack overflow, etc). Make sure to use the linkage conventions applicable to the function you are debugging. For more information, see the *OS/390 V2R10 Language Environment Debugging Guide and Run-Time Messages*, SC28-1942.

#### 3.4.1 Debugger support

Both the dbx and Debug Tool debuggers provide seamless debugging of XPLink applications. That is, they handle transitions between upward-growing and downward-growing stack frames, the differences in stack pointers, etc. It is still the responsibility of the person doing the debug work to recognize the linkage conventions of the function being examined with the debugger.

The XPLink support in dbx ships as part of OS/390 V2R10. The XPLink support in Debug Tool requires the installation of PTFs for APARS PQ33668 and PQ37039.

See Appendix B, “Sample CEEDUMP header” on page 75 for a CEEDUMP traceback.

---

### 3.5 Tracing an XPLink application

The Performance Analyzer, which is one of the tools in the IBM C/C++ Productivity Tools for OS/390 (product number 5655-B85), traces, or profiles, the execution of your program on the host and creates a trace file that can be analyzed in several diagrams at the workstation.

The Performance Analyzer provide seamless tracing of XPLink applications. That is, it handles transitions between upward-growing and downward-growing stack frames, the differences in stack pointers, etc.

The XPLink support in the Performance Analyzer requires the installation of PTF UQ47678. Installation of the LE PTFs UQ44252, UQ43444 and UQ47487 is also required.

---

## 3.6 Storage tuning

In XPLink, tuning the stack allocation is extremely important. The following three approaches are available:

1. Recompile to determine stack requirements
2. Estimate stack requirements
3. Unknowable stack requirements

For information on further storage tuning refer to the redbook *Tuning Large C/C++ Applications On OS/390 UNIX System Services*, SG24-5606.

### 3.6.1 Recompile to determine stack requirements

In the first approach, you start by recompiling the entire application specifying XPLINK(NOGUARD). This forces the compiler to generate function prologs that do not rely on the guard page to check for stack overflow. These are the steps:

1. Compile the entire application specifying XPLINK(NOGUARD).
2. Specify the RPTSTG(ON) and RPTOPTS(ON) run-time options.
3. Run the application using test data that will result in the greatest stack allocation, including `alloca()` use.
4. Examine the Storage Report produced by LE to determine the recommended stack size.
5. Optionally, specify the recommended stack size in a `#pragma runopts` directive in the application source.
6. Recompile the application specifying XPLINK(GUARD).
7. Run the application using the recommended stack size (STACK run-time option).

### 3.6.2 Estimate stack requirements

In the second approach, you need to make an estimate of the stack requirements of the application, and run the application with an initial stack allocation that is *certain* to exceed this estimate. Failure to estimate a large enough initial allocation can result in test runs that consume excessive system resources. The advantage of this approach is that you don't need to determine the best stack size; the disadvantage is that you need a good estimate of the stack requirements before beginning. These are the steps:

1. Specify the RPTSTG(ON), RPTOPTS(ON), and STACK run-time options, ensuring an adequate initial XPLink stack allocation in the STACK option.
2. Run the application using test data that will result in the greatest stack allocation, including `alloca()` use.
3. Examine the Storage Report produced by LE to determine the recommended stack size.
4. Optionally, specify the recommended stack size in a `#pragma runopts` directive in the application source.
5. Run the application using the recommended stack size.

### 3.6.3 Unknowable stack requirements

The third approach is required when you are unable to determine the stack requirements of the application. This could happen in a number of circumstances, including: the application might be in some way recursive, might use `alloca()` with storage amounts computed at run time, or might use XPLink DLLs from other sources that may change after application deployment. In this approach, you sacrifice some of the XPLink performance advantage in at least parts of the application to avoid the disastrous run-time penalty resulting from repeated attempts to store into the guard page when running out of stack space. Here are the steps to follow:

1. Compile the entire application specifying `XPLINK(NOGUARD)`.
2. Specify the `RPTSTG(ON)`, `RPTOPTS(ON)`, and `STACK` run-time options, ensuring an adequate initial XPLink stack allocation in the `STACK` option.
3. Run the application using test data that you think will be typical in its use of the stack.
4. Examine the Storage Report produced by LE to determine the recommended stack size.
5. Optionally, recompile parts of the application that you *know* will be run only within the initial stack allocation (`main()`, for example) specifying `XPLINK(GUARD)`.
6. Run the application using the recommended stack size (`STACK` run-time option) and the `RPTSTG(ON)` run-time option.
7. Monitor application use of stack storage to determine changes in the application's typical behavior.
8. Change the `STACK` run-time option if suggested repeatedly by the LE storage report.

---

## Chapter 4. Compatibility considerations

An application will receive the greatest performance improvement by recompiling all of its functions as XPLink. Unfortunately, other considerations often come into play that prevent this from being possible. For example, maybe your application consists partly of routines written in COBOL or PL/I, which don't support XPLink. Or you may have some S/390 assembler code and rewriting it using XPLink coding conventions is not feasible. Or possibly your application is using a third-party product that is shipped in a DLL and there is no XPLink version available. In these cases, your application will consist of both XPLink and non-XPLink functions which must be able to communicate with each other.

---

### 4.1 Mixing XPLink and non-XPLink code

The basic mechanism provided to allow XPLink and non-XPLink functions to call each other is the use of a Dynamic Link Library (DLL). In general, it is not possible to mix XPLink and non-XPLink functions in the same program object. However, a non-XPLink function in a DLL application can call an XPLink function in a separate DLL, and vice versa. In this case, LE will insert some "glue code" between the caller and called function to convert between the incompatible linkage conventions. There is a significant cost associated with this glue code, shown in 5.4, "The cost of glue code" on page 45. Therefore, one of your first considerations when choosing an application that can only be partially converted to XPLink is to decide if the number of "pure" XPLink function calls (where a performance improvement will be gained) will recoup the glue code costs of cross-linkage calls.

Some things to consider:

- There are two versions of the LE C/C++ Run-time Library (RTL). One version is non-XPLink, used when it is known there is no requirement for XPLink support. This is the same RTL that existed prior to the XPLink support. No XPLink resources are allocated, so there is no extra overhead due to XPLink support. All calls to this RTL are made using non-XPLink conventions.

This non-XPLink LE C/C++ RTL exists in the CEEEV003 member of the CEE.SCEERUN data set. References to its functions and data are resolved statically through the SCEELKED/SCEELKEX/SCEE OBJ/SCEECPP data sets. Refer to 3.1, "Installation and customization" on page 13 for more details.

- The second version of the LE C/C++ RTL is the XPLink version. It contains support for both XPLink and non-XPLink callers. Most calls to this RTL are made using XPLink conventions. If there are non-XPLink callers of the XPLink C/C++ RTL, then the necessary glue code will be inserted to convert to XPLink conventions. Therefore, the number of calls from non-XPLink functions to the XPLink C/C++ RTL must be considered when weighing the costs associated with moving an application to XPLink.

In OS/390 Release 10, the XPLink LE C/C++ RTL itself is also a mixed XPLink and non-XPLink application, so internally it may make some cross linkage calls that will have glue code costs associated with them. For this reason, *applications that are heavy users of the C/C++ RTL may not currently be good candidates for conversion to XPLink, but each application should be evaluated individually.* We measured some of the effects of this, as described in 5.6, "The cost of calling the C run-time library" on page 49.

This XPLink LE C/C++ RTL exists in the CELHV003 member of the CEE.SCEERUN2 data set. It is a DLL, so references to its functions and data are resolved dynamically through the CELHS003 side deck in the CEE.SCEELIB data set. There is also a CELHS001 side deck for CEL AWIs, and a CELHSCPP side deck for C++ functions and operators. Refer to 3.1, “Installation and customization” on page 13 for more details.

- One of the resources of the XPLink C/C++ RTL that is not called using XPLink conventions is the Hex Floating Point (HFP) Math Library. All HFP math functions are always called using non-XPLink conventions, so XPLink callers will go through glue code to reach these functions. However, the Binary Floating Point (BFP) Math Library is compiled XPLink in the XPLink version of the RTL, so there are no glue code costs for XPLink callers in this case. For this reason, *applications that use the BFP Math Library may be better suited to XPLink than those that use the HFP Math Library*. This is shown in Figure 16 on page 61.

Besides the DLL call mechanism, another way to mix XPLink and non-XPLink functions is through the use of the C fetch() service. A non-XPLink function may fetch() an XPLink module, and vice versa. Note that an XPLink module that is going to be fetched *must* have its entry point specified with the `#pragma(..., fetchable)` preprocessor directive.

The following call mechanisms, which support calls across load module and program object boundaries, *do not* support calls across linkage types. They do not have support for inserting the necessary glue code:

- COBOL Dynamic Call
- PL/I FETCH
- Traditional S/390 LOAD and BALR

---

## 4.2 Mixing assembler code with XPLink

You must first consider whether the assembler code can be converted, or written, to XPLink conventions.

### 4.2.1 XPLink assembler

S/390 High Level Assembler routines can be coded to use XPLink conventions. The intent of the XPLink assembler support is to be able to make high performance calls to small assembler routines to perform specific functions that are not easily accomplished in C or C++. With this in mind, *in OS/390 V2R10, XPLink Assembler routines are restricted to be leaf routines. This means that they cannot make calls to other functions*. There are two reasons for this restriction:

- In order for an XPLink (reentrant) function to call another function, it must be able to create a reference to that function in the Writeable Static Area (WSA). In non-XPLink code this reference is typically a V-con to the function's entry point. In XPLink code it is a function descriptor containing addresses of both the function's environment and entry point. There is currently no mechanism in the assembler to support creating these function descriptors in WSA.
- There is no XPLink CALL macro support. The XPLink calling conventions are more complex than non-XPLink calling conventions. While the XPLink calling



conventions are described in *OS/390 V2R10.0 Language Environment Vendor Interfaces*, SY28-1152, they are not for the faint of heart.

It is IBM's intent to lift this restriction in a subsequent release. For the time being, the following work-arounds can be used:

- Write the XPLink assembler function as non-reentrant. This will create the function descriptor as part of the executable, instead of placing it in WSA. This works if the XPLink function being called is going to be statically bound as part of the program object.
- Take the address of the function to be called in C or C++ code, then pass this function pointer to the XPLink Assembler code and make a call using it.

Both of these work-arounds require some knowledge of the mechanics involved in making an XPLink call, which can be found in *OS/390 V2R10.0 Language Environment Vendor Interfaces*.

#### 4.2.1.1 XPLink assembler macro support

The following assembler macros are provided by Language Environment in the CEE.SCEEMAC data set to facilitate creating XPLink assembler routines:

**EDCXPRLG:** Macro to create XPLink-conforming function prolog code. It has the following keyword parameters:

- **DSASIZE=ddd** (required): where **ddd** is the size, in decimal, of the stack frame to buy. The term “buy” implies that the stack register (GPR4) will be decremented by the size of the stack frame. The prolog code will also verify that this new stack lies within the current stack segment. If it doesn't, an overflow has occurred and a new stack segment will automatically be obtained and initialized.

Note that if **ddd** is larger than 2K, then GT2KSTK=YES must also be specified.

DSASIZE can be specified as zero (DSASIZE=0), but then no stack storage can be used. This also means you cannot save and restore registers because the code that checks for stack overflow has not been generated. If DSASIZE is specified, the minimum value must be 80 (decimal).

- **PARMWRDS=ddd** (optional, default is PARMWRDS=0): This parameter is not strictly necessary, but will help generate more efficient Program Prolog Area (PPA) structures that will speed up stack overflow (if it occurs while calling this function). It should be specified as the (decimal) number of 4-byte words contained in the input parameter list. Since the parameter list is contained in the caller's stack frame, if a stack overflow occurs, this portion of the caller's stack frame must be copied to the “transitional” stack frame in the new stack segment.

If this parameter is omitted, the function will be marked as taking a variable number of arguments.

- **ENTNAME=label** (optional): If no entry point name is provided or the entry point name is the same as the macro label, then the label on the macro will be used to name the XPLink function entry point, and a new label is constructed to name the XPLink entry point marker by appending a '#C' to the macro label. Otherwise the XPLink entry point

marker is set to the macro label and the function entry is set to the provided entry point name.

- **BASEREG=dd** or **BASEREG=NONE** (optional, default is **BASEREG=8**)  
If **NONE** is specified, a base register will not be established. If **dd** is specified, that decimal number will be used to indicate the register to be established as the base register. Note that a valid **DSASIZE** must be specified in this case or the previous contents of the base register will not be preserved across the function call.
- **PSECT=label** (optional): This is the label name to be assigned to the XPLink assembler function **PSECT** area. This can be used to establish an XPLink environment for the assembler function.

If a **PSECT** area is defined for an XPLink assembler function, then when that XPLink assembler function is called, **GPR5** will contain the address of its **PSECT** area (this is part of the XPLink calling conventions).

- **GT2KSTK=YES** (optional, default is **NO**): If **YES** is specified, then the (less efficient) prolog code to explicitly check for stack overflow will be generated. If the value of **DSASIZE** specified is greater than 2048, then **GT2KSTK=YES** must be specified.

**EDCXEPLG:** Macro to create XPLink-conforming function epilog code. It has no parameters. Besides creating function epilog code, it will also generate a standard form of the necessary PPA control blocks.

**CEEDSA:** Macro to produce a DSA mapping. It will produce a non-XPLink DSA mapping, an XPLink DSA mapping, or both.

#### 4.2.1.2 XPLink Assembler coding conventions

In order to create S/390 assembler code that meets XPLink conventions, the following changes from OS Linkage conventions used in traditional assembler code must be recognized:

- The XPLink stack is downward-growing and protected by a 4K guard page. The **EDCXPRLG** and **EDCXEPLG** macros will generate the necessary code to create XPLink function prologs and epilogs.
- The XPLink stack register, **GPR4**, is “biased”. The term biased means the address in **GPR4** is actually 2K (2048) bytes *before* the actual start of the current stack frame. The advantage to doing this is that for small stack frames (less than 2K), the address generation interlock (AGI) which occurs when the stack pointer is updated is moved to a position where it is more likely that it can be filled by an optimizer. See “Glossary” on page 83.

In addition, XPLink leaf (Xleaf) routines may not “buy” a stack frame. In this case Xleaf prolog code does not decrement **GPR4**, and it continues to point to 2K before the start of the caller’s stack frame.

- The format of the stack frame has changed for XPLink. Refer to Figure 2 on page 9 for the new format. One of the most important aspects of this change is that *a called function will find its input parameters in some combination of registers and the argument area in the caller’s stack frame.*

The argument area in the stack frame represents the largest parameter list on all calls made by that function. If a parameter is passed in a register (general purpose or floating point), then the contents of that parameter’s location in the argument area are undetermined. If the **STOREARGS** suboption of the XPLink compile option is specified, then a called function will ensure that the

argument area in its caller's stack frame is completely filled in. This can be an aid during debugging when trying to locate input parameters, but it will adversely affect the performance of that function's prolog.

- XPLink register conventions. These are compared to non-XPLink register conventions in Table 2.

Table 2. Comparison of non-XPLink and XPLink register conventions

	non-XPLink	XPLink
Stack Pointer	GPR13	GPR4 (biased)
Return Address	GPR14	GPR7
Entry Point Address on function entry	GPR15	GPR6 (not guaranteed; a routine may be called via relative branch)
Environment	GPR0 <sup>1</sup> (Writeable Static Address)	GPR5
CAA Address	GPR12 <sup>1</sup>	GPR12
Input Parameter List	address in GPR1	Located at fixed offset 64 (x'40') into the caller's DSA (remember the 2K bias on GPR4). Additionally, any of GPRs 1, 2, and 3, and FPRs 0, 2, 4, and 6, may be used to pass parameters instead of the caller's stack frame
Return Code	GPR15	GPR3 (extended return value in GPRs 1-2)
Start address of called function's DSA	Next Available Byte (NAB) in caller's DSA <sup>1</sup>	Caller's GPR4 minus called function's DSA size
End address of called function's DSA	NAB in caller's DSA plus called function's DSA size <sup>1</sup>	Caller's GPR4
Where caller's registers are saved	GPRs 0-12 saved in caller's DSA GPR13 saved in called function's DSA <sup>1</sup> GPRs 14-15 saved in caller's DSA	GPR0, not saved, not preserved GPRs 1-3, not saved, not preserved GPR4, not saved, recalculated (or saved, restored) GPR5, not saved, not preserved GPR6 saved in called function's stack frame, not restored GPRs 7-15 saved in called function's stack frame (GPR7 is the return register and is not guaranteed to be restored)

1. Only present as part of Standard LE Linkage (which is a superset of OS Linkage).

#### 4.2.1.3 XPLink Assembler example 1

The following sample XPLink code uses compare-and-swap (CS) to attempt to atomically set the value of the specified variable to its current value plus 1. If the value of the variable after the CS is not what we expect it to be, then the current value of the variable is reloaded and the CS is tried again.

```
*
* AtomicIncrement - atomically increment the value of a variable
*
```

```

* Inputs: R1 - location of the variable to increment
*
* Outputs: the incremented value of the variable
*
ATMCINCT TITLE 'Atomic Increment'
ATOMCINC EDCXPRLG DSASIZE=0,BASEREG=NONE
*
        L      R0,0(,R1)      load current value into R0
ATMCIRTY DS      0H
        LR      R3,R0         copy current value to R3
        AHI      R3,1         increment the variable
        CS      R0,R3,0(R1)   try to store incr value
        JNE      ATMCIRTY     retry if necessary
*
        EDCXEPLG             return
*
R0      EQU      0
R1      EQU      1
R2      EQU      2
R3      EQU      3
END

```

There are a several points worth noting in this example:

- The EDCXPRLG macro is used to generate XPLink prolog code, including an XPLink entry marker. Since there is no requirement for automatic data in this function, DSASIZE=0 was specified. If an XPLink assembler function needs automatic data, even if it's just to save and restore a register so it can be used as a work register, you should specify a minimum of DSASIZE=80 (that's 80 decimal). This will cover the minimum areas of the XPLink save area. It will also make EDCXPRLG generate the necessary STM instruction that will:
  - a. Save (and later restore) the non-volatile registers.
  - b. Check for stack overflow. Early in program check handling LE will explicitly look for an STM beginning with GPRs 4, 5, or 6, into the guard page (among other things), to determine if a stack overflow occurred.
- The EDCXEPLG macro is used to generate XPLink epilog code, including the LE Program Prolog Area (PPA) structures. When DSASIZE=0 is specified on the EDCXPRLG macro, EDCXEPLG will *not* generate code to restore the non-volatile registers from the save area.
- Since XPLink will pass up to the first 3 “integral” parameters in GPRs 1 through 3, the input parameter (the address of the variable to increment) is directly accessible. This saves an extra level of indirection and therefore several instructions that would otherwise be necessary to set up and access the parameter list.
- This XPLink assembler function uses GPRs 0 and 3 as work registers, as these are volatile registers and don't need to be preserved across the call.
- Not directly XPLink related, but this function uses a Branch Relative instruction, so it can avoid setting up and using a base register.

#### 4.2.2 Non-XPLink assembler

When the Binder is used to create an XPLink executable, all the objects that are bound together must be XPLink-compiled objects. There are two reasons for this:

- There is a cost to performance for the glue code associated with switching between XPLink and non-XPLink linkages. Since it is highly recommended that an application limit the use of glue code, it was decided to limit cross linkage calls to those at the program object boundary (for example, calls between DLLs). LE will insert the necessary glue code on supported cross linkage calls across a program object boundary, but the Binder has no such facility.
- Traditional non-XPLink calls to LE and C/C++ RTL services are resolved statically through “stubs” that exist in the SCEELKED, SCEELKEX, SCEEOBJ, and SCEECPP data sets. However, the use of stubs in the XPLink version of LE has been greatly reduced. All XPLink calls to LE and C/C++ RTL services are resolved dynamically using the side decks available in the SCEELIB data set (there is a minimal set of stubs in the SCEEBIND data set, but these don’t represent interfaces to LE and C/C++ RTL services). Because the Binder resolves all static references first, *before* performing any dynamic resolution, you cannot specify both the SCEELKED group of data sets at the same time that you would specify the SCEELIB side decks. If you did, XPLink references to LE and C/C++ RTL services would be resolved statically using the non-XPLink stubs, and the Binder would detect a linkage mismatch and issue the IEW2469E message (see 3.2.2, “Binder support” on page 16).

However, there is one opportunity to allow mixing XPLink and non-XPLink objects in the same program object that will make it easier to port applications to XPLink. That is, calls from XPLink functions to OS Linkage routines *can* occur within the same program object.

The rationale for doing this is twofold:

- Interfaces to OS/390 system services are typically pure OS Linkage. These calls can be made using the OS\_NOSTACK call mechanism. In this case there is no compiler-inserted glue code. Instead, the compiler will emulate OS Linkage calling conventions, also providing a 72-byte save area that the called function can use to store the caller’s registers in.
- Since the effort to port Assembler functions from non-XPLink to XPLink can be significant, certain Assembler functions can remain non-XPLink and be called using the OS\_UPSTACK mechanism. Note that this should not be done for Assembler functions that are performance-sensitive, because a call to compiler-inserted glue code will occur. In this case, these functions should be rewritten using XPLink conventions. However, Assembler functions that are sensitive to the performance overhead of the function call linkage are usually small in size, and the rewrite in this case should not be significant.

#### 4.2.2.1 Specifying OS Linkage calls from C/C++

The `#pragma linkage C` directive can be used to specify to the compiler that a call to a function should use OS Linkage calling conventions. For more details, refer to the `#pragma linkage` directive in the *OS/390 C/C++ Language Reference*, SC09-2360.

In C++, you can use the extern storage class specifier to specify that a called function is to use OS Linkage calling conventions. For more details, refer to the extern storage class specifier in the *OS/390 C/C++ Language Reference*.

#### 4.2.2.2 OS\_NOSTACK calls

A called function that is specified as OS\_NOSTACK will be called using OS Linkage conventions. Specifically:

- The parameter list will be an OS Linkage parameter list, with register 1 containing the address of a list of addresses of parameters. The high order bit of the address of the last parameter will be turned on.
- Register 13 will contain the address of a 72-byte save area that the called function can use to store the caller's registers into.
- The function will be called using a BALR 14,15 instruction, where register 15 contains the address of the executable entry point of the called function, and register 14 will contain the executable return point into the caller.

Typically, OS\_NOSTACK is used to call a stub routine that will save the caller's registers and then PC or SVC to a system service. However, any OS Linkage routine that has no stack dependencies other than a 72-byte save area can be defined as OS\_NOSTACK.

The compiler will emulate the OS Linkage call inline, so there is no requirement for a call to glue code to convert from XPLink to non-XPLink calling conventions. However, by its nature the OS Linkage call will not be as efficient as a pure XPLink call, so if performance is an issue the called routine should be converted to XPLink.

For example, a function `foo()` can be specified as OS\_NOSTACK using:

```
#pragma linkage(foo, OS_NOSTACK) /* from an XPLink C caller */
extern "OS_NOSTACK" foo;         /* from an XPLink C++ caller */
```

Additionally, OS\_NOSTACK is the default OS Linkage used for XPLink-compiled functions. So, if you didn't override the default OS Linkage convention using the OSCALL suboption (see below), then OS\_NOSTACK will also be assumed for:

```
#pragma linkage(foo, OS) /* defaults to OS_NOSTACK from XPLink C caller */
extern "OS" foo;         /* defaults to OS_NOSTACK from XPLink C++ caller */
```

#### 4.2.2.3 OS\_UPSTACK calls

A called function that is specified as OS\_UPSTACK will be called using Standard LE Linkage conventions. Standard LE Linkage conventions include everything that OS Linkage specifies, and in addition:

- Register 12 will contain the address of the CAA, an LE control block.
- Register 13 will contain the address of an LE-conforming stack frame. This includes an area for saving the caller's registers, plus a Next Available Byte (NAB) field, which contains the address following the end of the current stack frame in the upward-growing stack. This allows a caller to allocate a stack frame at this position.
- The entry point is "LE-conforming". This includes Program Prolog Area (PPA) control blocks that describe the function.

In order to accomplish the cross-linkage call, the compiler will emit a call to the glue routine `@@D2U@OS`. The glue routine will be passed the address of the target OS\_UPSTACK routine. In order to make this call, the glue routine must do the following:

- Switch from the downward-growing XPLink stack to the upward-growing non-XPLink stack, with the necessary links between the two.
- Convert from XPLink register conventions to non-XPLink register conventions.
- Convert from XPLink parameter passing conventions to OS Linkage parameter passing conventions. The parameter list is already set up as a logical “by reference” OS Linkage parameter list. That is, it is built as a list of addresses of parameters, with the high order bit of the last parameter’s address turned on. However, the XPLink caller uses XPLink parameter passing “mechanics”, so up to the first three parameter addresses will be passed in registers. The OS\_UPSTACK glue code will convert this so that register 1 points to the complete list of addresses.
- Call the target OS\_UPSTACK routine using OS Linkage conventions (i.e. BALR 14,15).
- When the target routine returns, the glue code must revert back to the downward-growing stack and XPLink register conventions. Additionally, the return value must be changed to use XPLink return value conventions.

Because of all the extra processing that the glue code must perform, an OS\_UPSTACK call has considerably more overhead than an OS\_NOSTACK call. If performance is an issue, you should consider converting the OS Linkage routine to XPLink, as described in 4.2.1, “XPLink assembler” on page 24.

A function `bar()` can be specified as OS\_UPSTACK using:

```
#pragma linkage(bar, OS_UPSTACK) /* from an XPLink C caller */
extern "OS_UPSTACK" bar;         /* from an XPLink C++ caller */
```

#### 4.2.2.4 OS\_DOWNSTACK calls

The OS\_DOWNSTACK call type provides a means to call an XPLink function using an OS Linkage “by reference” parameter list, but with XPLink parameter passing mechanics.

The parameter list will be built as a logical list of addresses pointing to the passed parameters. The high order bit of the last address will be turned on. However, instead of putting the address of the start of this list of parameter addresses into register 1, up to the first three parameter addresses will be passed in general registers 1, 2, and 3. Any remaining parameter addresses will be passed in the argument area in the caller’s stack frame. This is part of the mechanics of passing XPLink parameters.

The called OS\_DOWNSTACK function will receive control running on the downward-growing XPLink stack. There is no glue code involved in making this call.

A function `xpbar()` can be specified as OS\_DOWNSTACK using:

```
#pragma linkage(xpbar, OS_DOWNSTACK) /* from an XPLink C caller */
extern "OS_DOWNSTACK" xpbar;         /* from an XPLink C++ caller */
```

In XPLink mode, the REFERENCE linkage specifier is equivalent to OS\_DOWNSTACK. In non-XPLink mode, REFERENCE is equivalent to OS. Unlike the OS linkage specifier, REFERENCE is not affected by the OSCALL suboption of the XPLink compile option. If your application uses the OS linkage specifier to communicate between C and C++ functions with “by reference”

parameter lists, consider using REFERENCE instead of OS to make the source code portable between XPLink and non-XPLink.

#### 4.2.2.5 OSCALL suboption of the XPLink compiler option

An interface specified with `#pragma linkage(..., OS)` (or `extern "OS"` for C++) will default to `OS_NOSTACK` when compiled XPLink. The main reason for doing this is that existing C interfaces to system services (like those shipped by OS/390) will be defined in headers using `#pragma linkage(..., OS)`. Since these interfaces have no requirement for an LE-conforming stack, they can default to `OS_NOSTACK` for the best possible performance when called from XPLink applications.

You can override this default setting using the OSCALL suboption of the XPLink compile option. For example, if your application calls a number of LE-conforming Assembler routines that will not be converted to XPLink, and you don't want to modify all the existing `#pragma linkage(..., OS)` directives, you could change the default using `XPLINK(OSCALL(UPSTACK))`. But this will degrade the performance of any calls specified as `#pragma linkage(..., OS)` that do not require standard LE linkage conventions, by now requiring that they go through glue code.

---

### 4.3 Callback function considerations

A *callback function* is defined as a function pointer that is passed to a function with the intent of being called by the function to which it was passed. A common use for a callback function is as a user-supplied function that may be passed via function pointer to a function in a DLL. Note that the DLL could have been provided by a third-party vendor.

To understand the limitations of Callback Functions under XPLink, it is necessary to understand the three types of function pointers that currently exist:

- |              |   |
|--------------|---|
| NODLL style  | This is a pointer to a function whose address is taken in C code that has been compiled with the NODLL compiler option. This type of function pointer can only be called by other C code that has been compiled NODLL.  |
| DLL style    | This is a pointer to a function whose address is taken in C code that has been compiled with the DLL compiler option, or in any non-XPLink C++ code (all C++ code is "DLL enabled", see the Glossary). This type of function pointer can be called from C code that has been compiled with either the NODLL or DLL compiler options, but not the XPLink compiler option. It can also be called by any C++ code that has not been compiled with the XPLink option. |
| XPLink style | This is a pointer to a function whose address is taken in C or C++ code that has been compiled with the XPLink compiler option. It can be called from any C or C++ code, irrespective of how it was compiled.   |

In this sense, function pointers are downward compatible, but not upward compatible. So if the address of a function is taken in NODLL-compiled C code and stored into a function pointer which is then passed to either DLL or XPLink compiled code, an error will occur if that DLL or XPLink-compiled code tries to



make a call through that function pointer. However, if the address of a function is taken in XPLink-compiled C or C++ code and stored into a function pointer, that function pointer can be passed to any C or C++ code and be called from that code.

There may be cases where non-XPLink, DLL-compiled C code (or non-XPLink C++ code) needs to make a call through a function pointer, and it may not be known if that function pointer is NODLL or DLL style. In this case, the caller can be compiled with the DLL(CALLBACKANY) compiler option. This will result in a call to LE anytime a function pointer call is made in that compiled code. LE will ensure that a correct call is made to the function represented by that function pointer, regardless of whether that function pointer is NODLL-style or DLL-style. The disadvantage is that this adds considerable overhead to every function pointer call made in the code that was compiled DLL(CBA) (CBA being the abbreviated compiler option).

The CALLBACKANY suboption is not a supported suboption of the XPLink compiler option. (This was true at the time of writing, but may change.) Instead, the LE CWI `__bldxfd()` is provided that will selectively convert any supported function pointer into an XPLink-style function pointer that can then be called by any C or C++ code. This CWI is described in detail in *OS/390 Language Environment Vendor Interfaces*, SY28-1152-09 or later. There is a single special case where the C/C++ compiler will generate an implicit call to `__bldxfd()`:

For every parameter of function pointer type that is passed to an exported function that has been compiled XPLink, the compiler will generate an implicit call to `__bldxfd()` to ensure that function pointer can be called by the XPLink function. For example, if the following source code is compiled XPLink:

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

the compiler will generate an implicit call to `__bldxfd()` for parameters `fpParm1` and `fpParm3` so that these function pointers can be called without concern in the function `foo()` (or any functions it calls, passing these function pointers as parameters).

Since you cannot easily mix XPLink and non-XPLink code in the same program object, the normal way for non-XPLink code to call XPLink code is to make a call to a function in a DLL (i.e. an exported function). If the function pointer is passed as a defined parameter, the implicit conversion will take place and the function pointer can be called.

Note that the following cases will not result in an implicit call to `__bldxfd()`, so they may result in run-time errors if the function pointer is an unsupported style and is called from XPLink code:

- A global function pointer global (e.g. an imported data item).
- A function pointer contained within a structure that is either global (e.g. imported) or passed as a parameter.

In these cases, if the function pointer was set in NODLL-compiled or DLL-compiled C code (or non-XPLink C++), then a call made through this

function pointer by XPLink-compiled code will result in an error. To avoid this error, an application must do one of the following:

- Add the function pointer as a parameter to an exported function, passed as a function pointer type.
- Make an explicit call to `__blxdxfd()` to convert the function pointer into a format suitable for use by all linkage types.
- Ensure that all code taking the address of functions is compiled XPLink.
- Call a separate C "wrapper" function that is compiled DLL(CBA) and does nothing more than accept the function pointer as input, call the function represented by the function pointer, and return the results. Note that you cannot normally mix XPLink-compiled and non-XPLink-compiled (including DLL(CBA)) code in the same program object, so this "wrapper" function must either exist in a separate non-XPLink DLL, or be defined to the XPLink caller as `OS_UPSTACK` and define itself using `#pragma linkage(myname, OS)`. The latter case will result in an implicit call by the compiler to transition to the upward-growing stack and pass the parameter(s) using OS linkage.

---

## 4.4 XPLink restrictions

The following list identifies environments or subsystems that do not support XPLink applications. Refer to the *OS/390 V2R10.0 Language Environment for OS/390 & VM Programming Guide*, SC28-1939 for more details.

- In general, you cannot mix XPLink and non-XPLink functions in the same program object. The only exception to this is non-XPLink routines that are statically bound with an XPLink caller that has defined them as `OS_NOSTACK` or `OS_UPSTACK`.
- All executables in an XPLink environment must be LE-conforming. That is, they were compiled with an LE-conforming compiler, or they are LE-conforming Assembler.
- XPLink-compiled objects must be link-edited using the DFSMS Version 1 Release 6 Program Management Binder. You cannot use the prelinker. The binder output of an XPLink application is a program object in PM3 format, and must reside in either a PDSE or the Hierarchical File System (HFS). An XPLink application cannot reside in a PDS.
- The following are *not* supported mechanisms for calling XPLink functions:
  - COBOL Dynamic Call
  - PL/I Fetch
  - CEEPIPI `call_main`, `call_sub`, or `call_sub_addr`
  - CEELOAD
  - Traditional MVS load and BALR to call an XPLink function
- XPLink applications must run AMODE 31 (the ALL31 run-time option is forced ON in an XPLink environment). This also means that any non-XPLink application running in an XPLink environment must also run AMODE 31.
- You cannot use LE Preinitialization Services (PIPI) to initialize an XPLink environment.

- There is *no support* for XPLink applications running under CICS. The main reason is that CICS currently does not support the notion of a guard page.
- XPLink applications can invoke DB2 services using the EXEC SQL interface. However, XPLink applications *cannot* run as DB2 stored procedures. DB2 stored procedures run in an LE PAPI environment, which doesn't support XPLink.
- XPLink applications can invoke IMS services using the `ctdli()` interface. However, XPLink applications *cannot* run as IMS transactions.
- A nested (child) enclave must run with the same XPLink environment as its parent. For example, a program running in a non-XPLink environment cannot issue a `system()` service to execute an XPLink program. If that same program specified the XPLink(ON) run-time option, it would then be able to use `system()` to execute the XPLink program.
- The CEEBXITA Assembler User Exit and the CEEBINT High Level Language (HLL) User Exit cannot be coded as XPLink functions. They must be non-XPLink and they currently cannot call C run-time library functions or functions in a DLL. Instead, consider using a C++ static constructor in place of the HLL User Exit.
- There is no support for XPLink as part of Library Routine Retention (LRR).
- You cannot use Run-Time Library Services (RTLS) in an XPLink environment.
- There is no support for System Programmer C (SPC) applications in an XPLink environment.
- There is no support for XPLink on VM.
- You cannot use the C Multitasking Facility (C MTF) in an XPLink environment.
- You cannot use PL/I Multitasking in an XPLink environment.
- You can run existing non-XPLink SOM applications in an XPLink environment, but you cannot create XPLink SOM applications.
- You cannot use the Distributed Computing Environment (DCE) in an XPLink environment.
- XPLink cannot be combined with Inter-Procedural Analysis (IPA) in C/C++.

---

## 4.5 OS/390 C/C++ Version 2 Release 10 enhancements

OS/390 C/C++ has made the following enhancements, in addition to XPLink, for OS/390 C/C++ Version 2 Release 10. For more information on these enhancements refer to the *OS/390 C/C++ User's Guide*, SC09-2361.

### 4.5.1 Generalized Object File Format (GOFF)

GOFF is the strategic object module format for S/390. It extends the capabilities of object modules to contain more information than current object modules. It removes the limitations of the previous object module format and supports future enhancements. GOFF makes re-binding easier and more efficient. It is required for XPLink. It is used by specifying the GOFF compiler option, and is also turned on when the XPLink compiler option is specified.

## 4.5.2 InterProcedural Analysis (IPA) level 2

Under IPA level 1, many optimizations, such as constant propagation and pointer analysis, are performed at the intraprocedural (subprogram) level. With IPA level 2, these optimizations are performed across the entire program, which can result in significant improvement in the generated code. IPA level 2 is turned on by specifying the IPA(LEVEL(2)) compiler option.

## 4.5.3 Addition of @STATIC map into compiler listing

This listing element is triggered by the XREF compiler option. It displays offset information for file scope read/write static variables.

For example, the static map produced for the following small C function:

```
static char* a[4];
char** func(void) {
    a[0]="string 0";
    a[2]="string 2";
    return a;
};
```

is:

```

                                * * * * *   S T A T I C   M A P   * * * * *
OFFSET (HEX)    LENGTH (HEX)    NAME
      0             9      ""1
      C             9      ""3
     18            10      a
                                * * * * *   E N D   O F   S T A T I C   M A P   * * * * *
```

The curious looking ""1 and ""3 entries in this map are for string literals; the default for C code being to put these in the WSA, making them read-write. When the ROSTRING compiler option is specified, the following writeable static map shows that the string literals have retreated to the code section where they, in the authors' opinion, belong:

```

                                * * * * *   S T A T I C   M A P   * * * * *
OFFSET (HEX)    LENGTH (HEX)    NAME
      0             10      a
                                * * * * *   E N D   O F   S T A T I C   M A P   * * * * *
```

## 4.5.4 COMPACT compiler option

The COMPACT compiler option is new. During optimizations performed during code generation, for both NOIPA and IPA, choices must be made between those optimizations which tend to result in faster but larger code and those which tend to result in smaller but slower code. The COMPACT | NOCOMPACT option controls these choices. When the COMPACT option is used, the compiler favors those optimizations which tend to limit the growth of the code. Because of the interaction between various optimizations, including inlining, code compiled with the COMPACT option may not always generate smaller code and data.

---

## 4.6 IPA and XPLink

IPA and XPLink address the issue of function call overhead in different ways. IPA can remove function call overhead by, for example, inlining functions where practical, removing the call completely. XPLink, on the other hand, reduces the cost of function calls without removing them.

C/C++/390 for OS/390 Version 2 Release 10 does not support the use of both IPA and XPLink in the program object.

In some cases XPLink will provide better application performance; in other cases, IPA will. In almost all cases, application build time will be lower with XPLink.



---

## Chapter 5. Performance characteristics

In this chapter we examine the effect of XPLink and its options on small fragments of code, such as function prologs and call sites, that XPLink and its options control. Typical code sequences resulting from some of these choices are shown in Appendix A, “XPLink code sequences” on page 73.

The code examples used for the measurements in this chapter are not part of any production system, nor indeed do they represent the “best” code or algorithm for their stated purposes. They are contrived solely for the purpose of illustrating XPLink performance characteristics.

---

### 5.1 Improvements in function prologs

XPLink, generally, is faster than traditional LE linkage conventions, and the primary source of this benefit is improved code sequences in function prologs. Clearly, there must be an upper bound to this improvement: we attempted to measure this by comparing the cost of function calls alone, as measured<sup>1</sup> using this code sequence:

```
void f20(void) {};  
void f19(void) { f20(); };  
void f18(void) { f19(); };  
void f17(void) { f18(); };  
void f16(void) { f17(); };  
void f15(void) { f16(); };  
void f14(void) { f15(); };  
void f13(void) { f14(); };  
void f12(void) { f13(); };  
void f11(void) { f12(); };  
void f10(void) { f11(); };  
void f9(void) { f10(); };  
void f8(void) { f9(); };  
void f7(void) { f8(); };  
void f6(void) { f7(); };  
void f5(void) { f6(); };  
void f4(void) { f5(); };  
void f3(void) { f4(); };  
void f2(void) { f3(); };  
void f1(void) { f2(); };  
void f(void) { f1(); };  
  
int main(void) {  
    int i;  
    startTimer();  
    for (i=0; i<bigNumber; ++i)  
        f();  
    endTimer();  
}
```

In this code, the leaf routine (f20) has no prolog at all in XPLink; the effect of this is somewhat masked by the call depth.

Figure 3 on page 40 shows the relative execution times for this loop (in this figure and in most figures in this book, the XPLink run is normalized to a value of 1 and

<sup>1</sup> Using the C run-time `clock()` function in a processor shared with other users

other runs are relative to it). This tells us that, at best, an application, when compiled with XPLink, will not run in less than approximately 40% of the time taken when compiled without XPLink (40% being the inverse of 2.5).

Of course, this is not a benefit that can be achieved by real applications, which do more than just call other meaningless functions; this test demonstrates the approximate upper bound of what can be achieved by rebuilding an application with XPLink. In fact, as we see later, some applications actually get slower when compiled with XPLink, although this situation is expected to be remedied to some degree in future releases. Furthermore, applications that mix XPLink and non-XPLink code may or may not be faster than an entirely non-XPLink version of the same application.

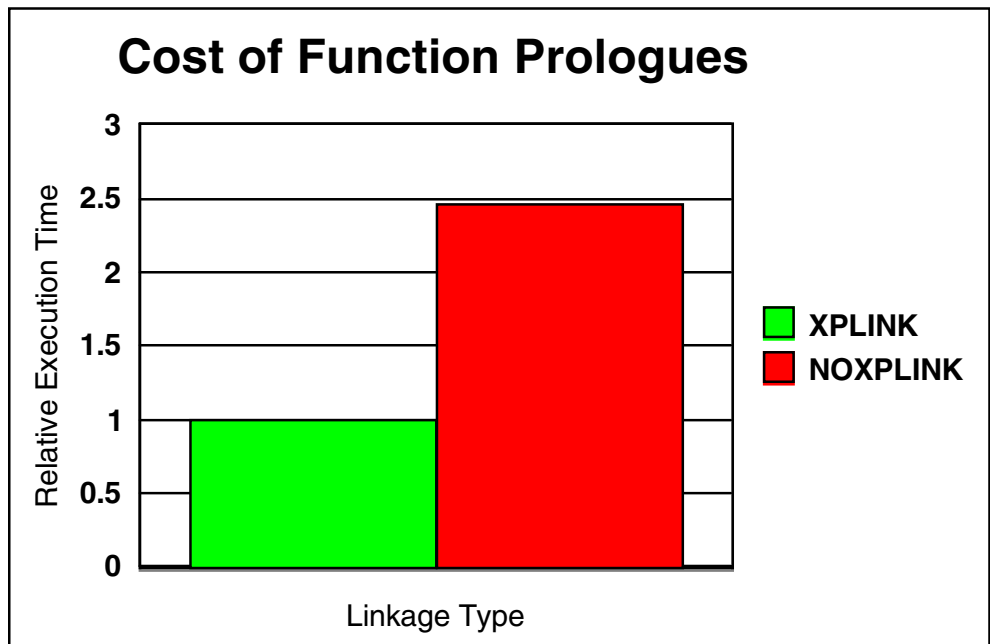


Figure 3. Cost of function prologs

## 5.2 The effect of stack size on XPLink performance

In both traditional OS/360 Linkage and its successor LE Linkage, ignoring the side effects of memory consumption, the cost of acquiring a stack frame is independent of the size of the stack frame.

This is not the case with XPLink, whose design intentionally favors functions having small stack frames (less than or equal to 2048 bytes), believed to be in the vast majority of most code, but especially in both object-oriented and C code. When using XPLink, different function-prolog code sequences are generated for functions having stack frames in the following four size ranges: 0 to 2048 bytes, 2049 to 4096 bytes, 4097 to 32768 bytes, and greater than 32768 bytes.

These differences arise from the following design choices and architectural characteristics:

- The choice to have the stack register point to 2048 bytes before the actual stack frame, allowing the Store Multiple instruction that saves the registers to be placed before the instruction that updates the stack pointer



- The choice of a 4096-byte guard page, forcing prolog code for functions with stack frames larger than 4096 bytes to explicitly check for stack overflow by comparing the proposed stack frame address with the stored end-of-stack field
- The 32768-byte limit on the value used in Add Halfword Immediate instructions

Examples of the prolog code generated by the C/C++ Compiler are shown in A.1, “Code sequences for function prologs” on page 73.

XPLink’s use of a “guard page” to detect stack overflow makes proper storage tuning of XPLink applications extremely important. This is discussed in 2.3.1.3, “Stack overflow detection” on page 7. In cases where storage tuning cannot be carried out, for example recursive applications or applications whose actual calling patterns cannot be known before the application is run, the compiler provides an XPLINK suboption, NOGUARD, to generate prologs that explicitly check for stack overflow by comparison with a known end-of-stack address, the strategy used in all cases for functions having stack frames larger than 4096 bytes.

Figure 4 on page 42 shows the approximate relative execution cost of a function call for the four stack frame size ranges and the three compiler options: XPLINK, XPLINK(NOGUARD), and NOXPLINK (the IBM-supplied default). The results shown in this section and the following section were computed using code with more “body”: a simulation (but not reporting) of the moves required in solving the “Towers of Hanoi” problem similar to this:

```
void move(int n,int currentPole,int usePole,int targetPole);
int main(void) {
    int i;
    startTimer();
    move(height,1,3,2);    // move height discs from post 1 to 2 via 3
    endTimer();
}
void move(int n,int currentPole,int usePole,int targetPole) {
    int i[useUpStackSpace]; // adjust to get different stack frame sizes
    if (n!=1){
        move(n-1,currentPole,targetPole,usePole);
        move(1,currentPole,usePole,targetPole);
        move(n-1,targetPole,usePole,currentPole);
    }
}
```

All tests were run with an adequate initial stack allocation.

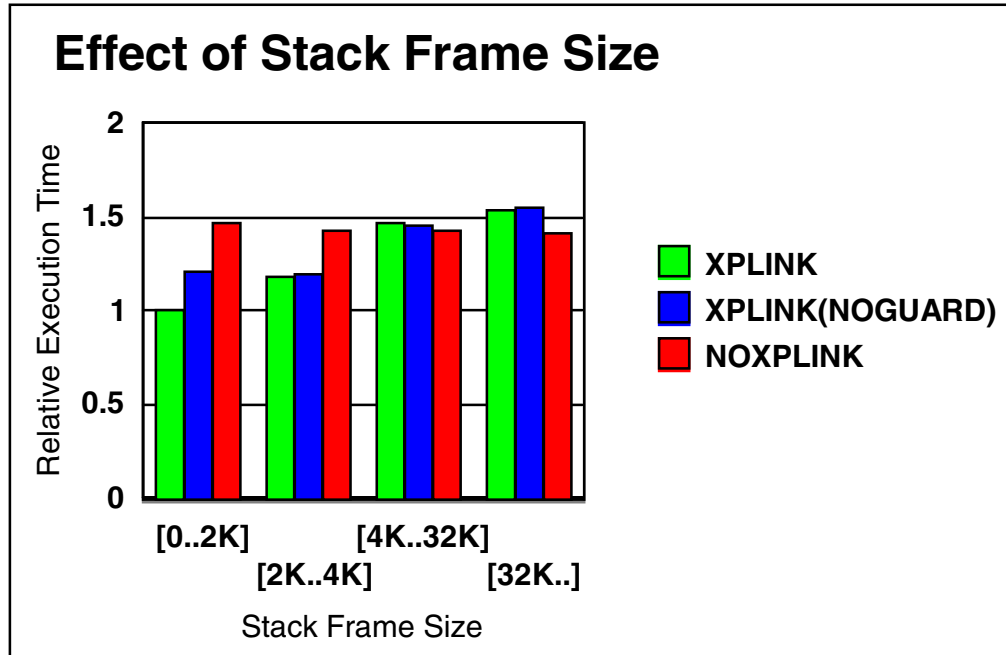


Figure 4. Effect of stack frame size on performance gain

Note that within the normally-expected range of error:

- All NOXPLINK times are the same.
- For stack frames larger than 4K, XPLINK and XPLINK(NOGUARD) are the same; identical code is generated by the compiler in this situation.
- For small stack frames, the XPLINK case uses approximately 70 % (1.0/1.43) of the time of the NOXPLINK case, a 30 % saving.

This example shows the one case where XPLink code is noticeably slower than non-XPLink code: functions with large stack frames (greater than 32768 bytes) *and* more than one argument are often marginally slower when compiled with XPLink. We expect future releases of the compiler will address this situation.

Note that functions having these characteristics can be freely mixed with other XPLink-compiled functions in the same program object, and that this penalty is incurred only for the functions having these characteristics, not the application as a whole.

### 5.2.1 The effect of parameter-list size

When this “application” is altered to pass only one parameter in its function calls (so this now simulates only the function call pattern) we see the comparisons shown in Figure 5 on page 43. In this case, the function prolog is no longer required to save work registers it uses to compute the new stack frame address and the performance penalty seen by functions with large stack frames disappears.

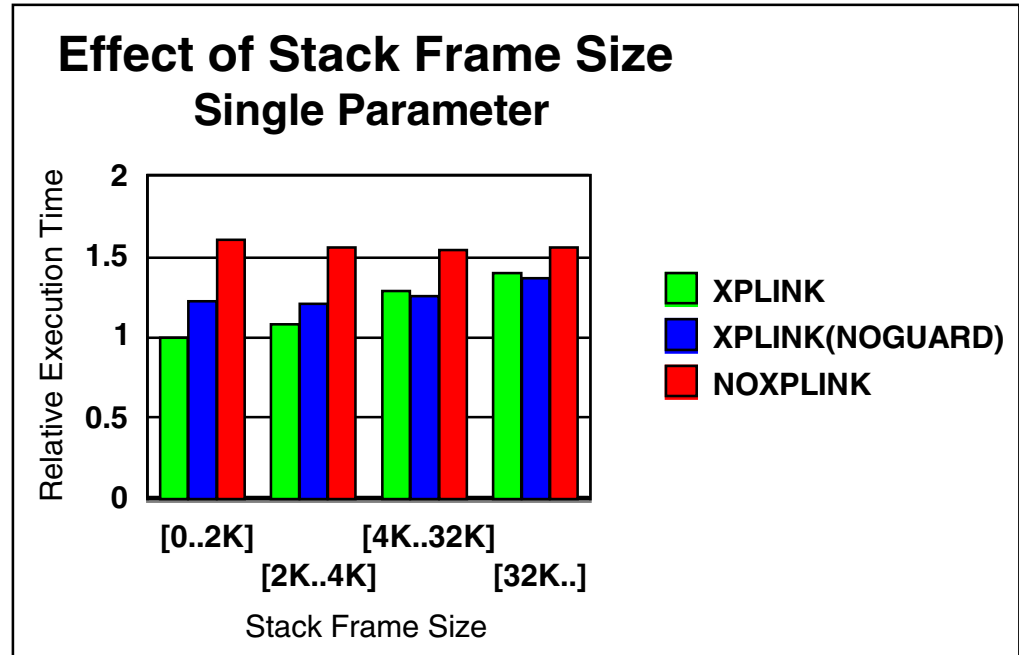


Figure 5. Effect of stack size on performance gain, single argument function calls

### 5.2.2 The effect of calling within a compilation unit

XPLink also makes improvements when the called function is in the same compilation unit as the caller. This benefit is realized at the call site by the removal of the requirement, in XPLink, that functions be entered with a valid base register.<sup>2</sup> The results of measuring this are shown in Figure 6 on page 44, where we see that the penalty associated with large stack frames is diminished.

<sup>2</sup> OS/390 Release 10 and XPLink require the "Immediate and Relative Branch Facility". Functions that use no read-only constants often require no code base register; they use relative branch instructions for any required control flow.

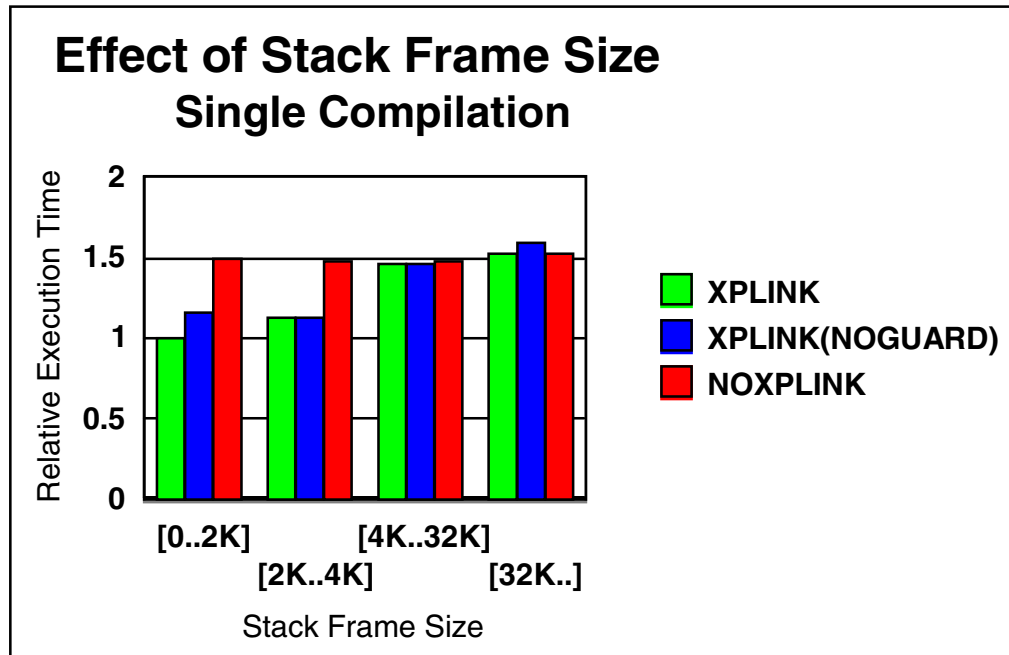


Figure 6. The effect of stack frame size on calling functions in the same compilation unit

### 5.3 The effect of XPLink suboptions on performance

Suboptions of the XPLink compiler option are available to control some aspects of the code generated for function prologs. These suboptions are:

- BACKCHAIN
- STOREARGS
- NOGUARD

NOGUARD is described in 2.3.1.3, “Stack overflow detection” on page 7; its performance implications are discussed in 5.2, “The effect of stack size on XPLink performance” on page 40. BACKCHAIN and STOREARGS are described in 2.3.1.2, “Static stack information” on page 6 and 2.3.3.2, “Arguments passed in registers” on page 10 respectively. The performance cost of these suboptions is shown in Figure 7. These suboptions do not depend on whether or not the function is in the same compilation unit as the caller so this was not measured. Their cost diminishes for functions with larger stack frames; the results shown are for small (less than 2048 bytes) stack frames.

Note that the effect of these suboptions on the overall application diminishes as the individual functions that comprise the application get larger. Furthermore, functions compiled with different XPLink suboptions may be freely mixed in an application; the performance penalty for using these options applies only when the functions using them are called.

The STOREARGS suboption provides increased, but not certain in the case of modified actual arguments, probability that debuggers may find the actual arguments passed to a function. BACKCHAIN is provided for the paranoid.

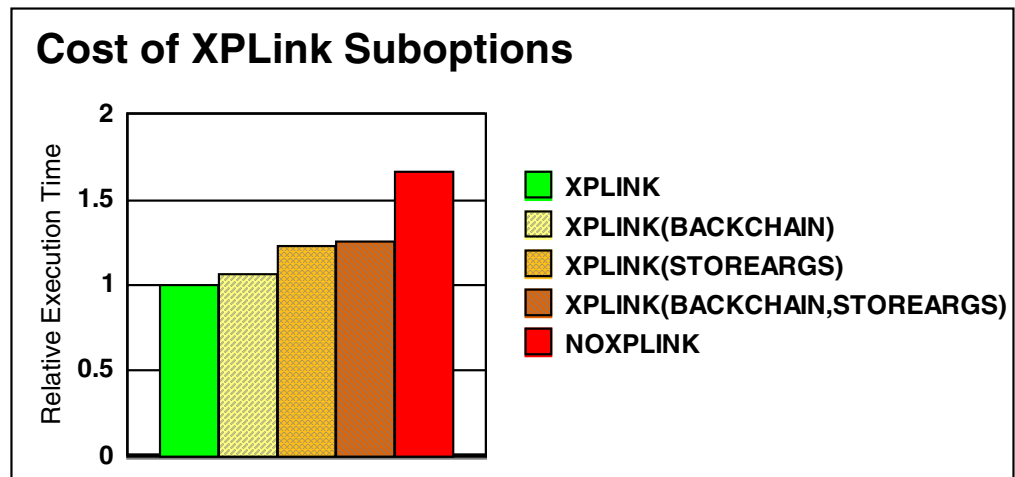


Figure 7. Cost of XPLink suboptions

## 5.4 The cost of glue code

Sometimes an application cannot be converted entirely to XPLink. This situation can arise, for example, when parts of the application are written in a language that doesn't support XPLink. The considerations relating to this are discussed in *OS/390 Language Environment for OS/390 & VM Writing Interlanguage Communication Applications*, SC28-1943-09 or later at:

<http://ibm.com/s390/bookmgr-cgi/bookmgr.exe/BOOKS/CEEA4030/2.0>

In this case, you should consider putting the code that can be recompiled and converted to XPLink into a separate DLL.

There are both costs and benefits associated with this. On the cost side, we see:

1. The effect of compiling the non-XPLink part of the application with the DLL option
2. The effect of the glue code that is called every time there is a control flow between the non-XPLink and XPLink parts of the application (discussed in 4.1, "Mixing XPLink and non-XPLink code" on page 23)

The benefits arise from the fast XPLink function calls within the XPLink portion.

The benefits are more likely to outweigh the costs when a single call into the XPLink portion of the application results in many function calls within the XPLink portion.

To illustrate this, we constructed an application in which a COBOL main program calls a recursive factorial function written in C. In this example, the two parts are easily separable (this won't always be the case in the real world) and the number of calls within the XPLink portion (the C code) can be easily varied by changing the factorial value to be computed.

We measured the cost of the DLL overhead and the transition glue code separately by first measuring the performance of the application after splitting it into a main program and DLL, and then measuring the performance after compiling the DLL with XPLink.

The results are shown in Figure 8. We see, in this example, a relatively constant cost (about 3 %) attributed to converting the application to a DLL architecture, although this cost diminishes very gradually as the proportion of time spent inside the DLL increases. We expect this value to vary according to the frequency of function calls in the main (non-XPLink) part of the application as well. We also see the combined effect when the DLL is converted to XPLink, varying from a cost of 65% when the DLL has only 4 internal calls for each transition into XPLink code (the worst ratio measured) to a benefit of nearly 40% when there are 49 internal calls for each transition into XPLink code. In this example the break-even point is when there are 22 calls for each transition; this will vary as the characteristics of the functions in the DLL change (number of parameters, size of stack frame, and so on): different applications will have different improvement potentials.

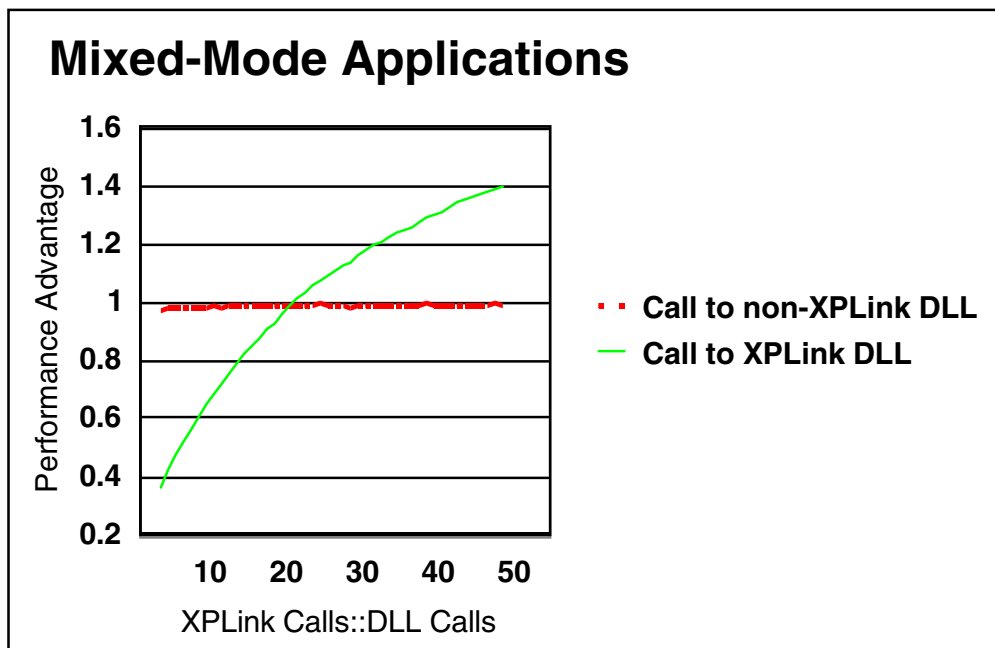


Figure 8. Benefit/cost of mixing XPLink and non-XPLink code

## 5.5 The cost of a mixed XPLink-assembler application

Many applications mix assembler language code with High Level Language (HLL) code. Unlike C or C++ code, converting assembler code to XPLink is not simply a matter of recompiling; the different register conventions and parameter-passing protocols mean that assembler code has to be modified or, in the worst case, completely rewritten to conform to XPLink. Furthermore, there is no facility in assembler that provides for access to Writeable Static data; applications that require this must be redesigned to pass addresses within the Writeable Static Area (WSA) as arguments to assembler code.

To help overcome these difficulties, the OS/390 C/C++ compiler offers linkage specifiers/pragmas that allow existing assembler code to be called, without change, from XPLink C or C++ code. These are documented in the *OS/390 C/C++ Programming Guide* (SC09-2362-06 or later) at

<http://ibm.com/s390/bookmgr-cgi/bookmgr.exe/BOOKS/CBCPG030/3.1.1>

If such an application is to be converted to XPLink, the user is left with the following choices, depending on the nature of the assembler code:

- Convert the assembler code to XPLink. Assembler macros are provided to generate XPLink prologs and epilogs, but not for XPLink calls. If the assembler code contains internal calls, either to other XPLink Assembler routines or to XPLink-enabled HLL code, the calls will have to be constructed by hand in assembler language. There are severe limitations on the support for reentrancy in this environment, discussed in “Mixing assembler code with XPLink” on page 24.

The effort to build XPLink call sequences by hand can vary from the nearly-trivial in the case where all arguments are integer types or are passed by reference, to the very-complicated and not for the faint of heart where the called function is a C/C++ vararg function expecting floating point parameters passed by value. The conventions for constructing argument lists are described in *OS/390 Language Environment for OS/390 & VM Vendor Interfaces*, SY28-1152-08 or later at:

<http://ibm.com/s390/bookmgr-cgi/bookmgr.exe/BOOKS/CEEV1000/2.4.3.5>

Of course, it's possible to convert only the outermost assembler code (the entry point called directly from XPLink C or C++ code) to XPLink, and use whatever “private” linkage convention (including linkage conventions using registers 13, 14, and 15) the author desires inside the assembler code.

Generally, this choice will yield the fastest execution time, especially if any internal assembler calls are converted to XPLink or a privately-defined XPLink-like style. The linkage specifier for the outer (called from XPLink code) assembler entry point should be “OS\_DOWNSTACK”.

- If the assembler code uses OS/360 Type 1 linkage for primary mode programs<sup>3</sup>, as described in 2.1, “History” on page 3 and in *OS/390 MVS Programming: Assembler Services Guide*, GC28-1762, then it can be called without change using the linkage specifier “OS\_NOSTACK”. This is the IBM-supplied default if the “OS” linkage specifier is used, but because this default can be changed on the command line used to invoke the compiler, the source of the calling program should be changed to specify “OS\_NOSTACK” or its synonym “OS31\_NOSTACK”.

Generally this will be marginally slower than converting the assembler code to XPLink.

- If the assembler code uses non-XPLink LE Linkage conventions<sup>4</sup>, as described in 2.1, “History” on page 3 and in *OS/390 Language Environment for OS/390 & VM Vendor Interfaces*, SY28-1152, the easiest alternative is to force the compiler to insert glue code similar to that used for XPLink to non-XPLink DLL transitions (as described in 5.4, “The cost of glue code” on page 45). This is done using the “OS\_UPSTACK” linkage specifier.

Generally, this is by far the most expensive in terms of execution time. The same considerations apply here as with the DLL case: the savings realized in the rest of the application by converting it to XPLink may or may not outweigh the costs of the glue code needed to call non-XPLink LE-conforming assembler code.

- Of course, the final alternative is to not convert the application to XPLink.

<sup>3</sup> That is, the assembler code does not use Access Register (AR) mode, or the linkage stack (as supported by the BAKR and PC instructions). Both are described in *Enterprise Systems Architecture/390 Principles of Operation*, SA22-7201.

<sup>4</sup> That is, the assembler code relies on the NAB or other LE-managed fields in the caller's stack frame or on the contents of the CAA.

Each of these four alternatives results in different instruction sequences for a function call. Because function calls appear in the bodies of functions, rather than in their prologs or epilogs, it's much harder to isolate the effects of these different code sequences. Many factors contribute to this, including the compiler's ability to move parts of the calling code sequence outside loops (an optimization that interferes with the style of code that is often used to measure performance) and the compiled code's use of general purpose registers in the vicinity of the call site.

We chose to measure four scenarios to illustrate a range of performance expectations. These were a main function containing a loop that:

1. Contained a call to an assembler function accepting no argument, which we called a "Direct call" in our measurements
2. Contained a call to a C function accepting no argument that in turn called this assembler function, which we called an "Indirect Call" in our measurements
3. Contained a call to an assembler function that accepted 3 arguments and returned their sum
4. Contained a call to a C function accepting 3 arguments that in turn called this assembler function passing in the 3 arguments and returned their sum

We tried to construct these scenarios in ways that minimized the beneficial effects of XPLink; of course the two "Indirect Call" cases do contain an XPLink C call inside the loop.

For each of these scenarios we measured the execution time for the three XPLink alternatives relative to the non-XPLink version of the code. We see mixed results, as shown in Figure 9. In three of the four cases, converting the assembler code to XPLink and using the "OS\_DOWNSTACK" linkage specifier was the fastest; in the remaining case, calling with the "OS\_NOSTACK" linkage specifier was the fastest. In all cases, the "OS\_UPSTACK" linkage specifier produced the worst code.

Using the "OS\_NOSTACK" specifier has the lowest conversion effort by far; we recommend this alternative for mixed XPLink-assembler applications unless there is a compelling performance advantage in converting the assembler code to XPLink.

Using "OS\_UPSTACK" should be considered only when the assembler code is an insignificant part of the application or cannot be converted from non-XPLink LE conventions to either XPLink or OS/360 Type 1 conventions. An application that is mostly LE-conforming assembler should not be converted to XPLink without carefully evaluating the cost of conversion and the risk of lost application reliability against the expected performance gains.



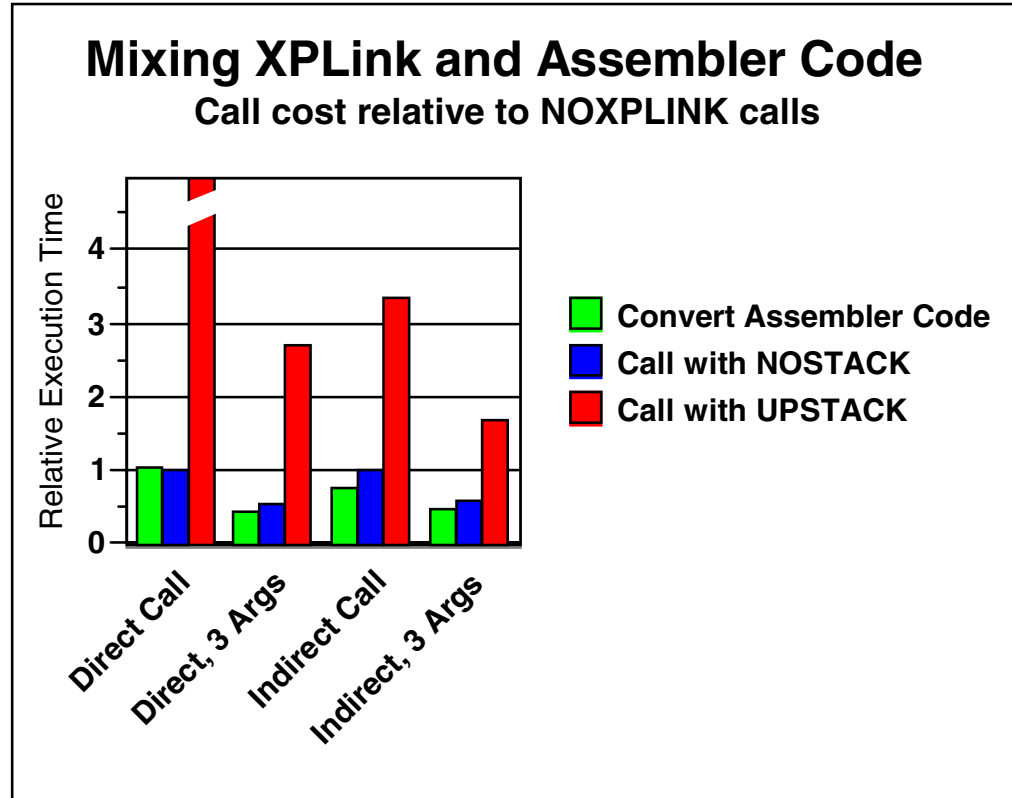


Figure 9. Mixing XPLink and assembler code

## 5.6 The cost of calling the C run-time library

In OS/390 Release 10, the XPLink C/C++ run-time library is a mixture of XPLink and non-XPLink code. Some of the implications of this are discussed in 4.1, “Mixing XPLink and non-XPLink code” on page 23.

It’s impossible to make a simple statement about the effect of calling the C run-time in an application because the complexity and internal structure of run-time functions vary greatly. In order to illustrate that this effect exists, we ran tests which made a fixed number of calls to a C run-time library function (`clock()`) and a varying number of XPLink function calls, with a varying number of arguments. The results are shown in Figure 10 on page 50.

As expected, when the ratio of XPLink calls to library calls was low, the overall performance of the test “application” degraded. As the number of XPLink calls was increased while holding the number of calls to the `clock()` function constant, the advantage of switching to XPLink became apparent. In this case, the point of trade-off varied between about 3 and 7 XPLink calls for every library call; your results will vary depending on the particular combination of run-time functions you call and the characteristics (number of arguments, stack frame size, and so on) of your XPLink calls.

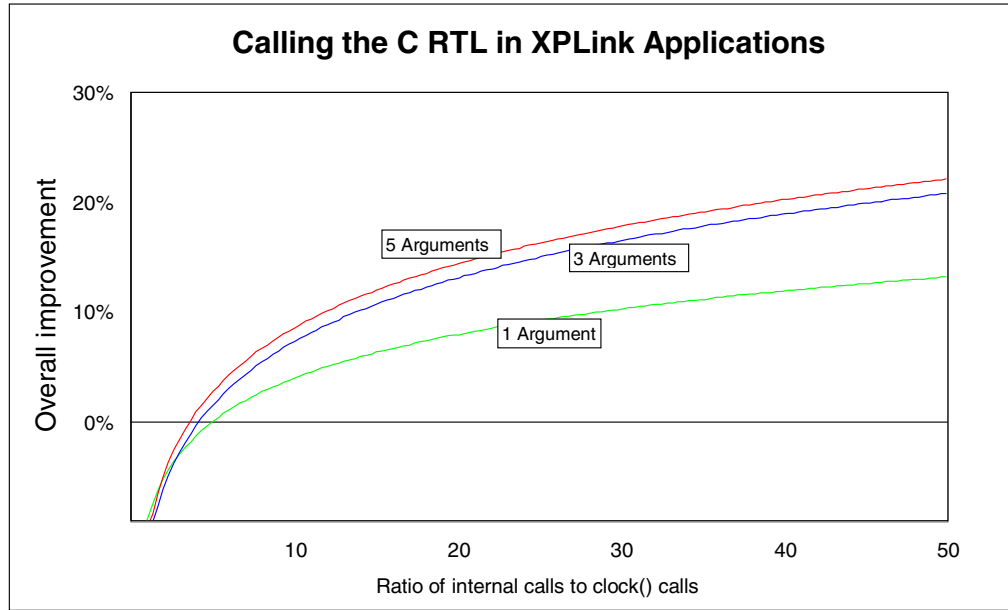


Figure 10. Calling the C run-time library in an XPLink application

## 5.7 XPLink and object-oriented code

Object-oriented code is often characterized by large numbers of small functions (methods) that do nothing more than set or retrieve state information in objects. Sometimes this involves simple computations involving the method's actual arguments, private data members, and static data; in many cases there's nothing more than setting or retrieving a private data member. Such functions are often "leaf routines", that is, they make no function call. The C++ compiler recognizes this and, for both XPLink and non-XPLink, is often able to suppress the generation of most of the code normally found in function prologs. This makes the XPLink and non-XPLink versions of such code very similar, although not identical due to the different register conventions. XPLink code may have an advantage when there are more than 2 arguments (in addition to the implied "this" pointer) or when the arguments include floating-point types; non-XPLink may have a slight advantage when the function uses literal constants. In either case, the performance characteristics of this style of code will be largely independent of the choice of linkage convention.

As the complexity of the methods increases, the advantages of XPLink resulting from the differences discussed in this chapter become dominant.

## 5.8 XPLink and C++ exception handling

XPLink achieves many of its performance goals by removing instructions that record information about the current stack state, substituting static control information. That requires much of the information about the current stack state to be computed, at some expense in execution time, rather than just "looked up" in storage. For example, by default, there is no direct link from an active stack frame to the stack frame of its caller; this must be computed by finding the function associated with every stack frame and, from there, control blocks that describe the stack frame.

We expect this design to cause facilities such as the C++ exception model (*throw-expressions*) to be somewhat slower; however, in many cases the benefit gained from improved function calling will more than outweigh the additional costs of `throw`. We were unable to measure this effect on the pre-GA OS/390 Release 10 system available to us in the summer of 2000.



---

## Chapter 6. Application and benchmark measurements

This chapter describes the results of measuring customer applications and development benchmarks with and without XPLink. It begins with a short review of measurement terminology and principles.

---

### 6.1 Quantification of the value of a system

Measurements provide important input toward quantifying the value of a given system. This chapter quantifies the *performance gains* possible for a variety of applications by measuring them with and without XPLink. Where appropriate, it also quantifies potential points of degradation.

Performance gain, of course, is described in units of time. Two formulations are used: time or its inverse, which is a rate. The former is called the response time or turnaround time of a transaction or application or job. It is an elapsed or “wall clock” time, or CPU time<sup>1</sup>. The inverse time or rate is called the throughput, or rate at which work is accomplished per unit time (*e.g.* transactions per second), where the work itself is really the average of some mix of work whose composition is well understood.

Both response time and rate measurement depend on the system configuration, and the background work that is concurrently executing (if any). Often response time is measured in a “dry” standalone environment, or in an artificial environment where the background work is strictly controlled. For example, response may be measured when 100 clients are all submitting the same transactions to the system. Or throughput may be measured by saturating the system with a particular, completely constrained mix of work.

Real system performance often depends on the ability of a system to handle much more varied workloads than in a “benchmark” environment, and often involves “ilities”, such as reliability, availability, securability, etc., which may actually degrade *benchmark performance* even while they enhance the value of the system. Thus it is important to keep in mind that the overall value of a system involves more than the quantification of a *single* metric, such as time, or even raw price/performance. One must be careful in not reading too much into the results of a single metric, such as performance gain.

For example, one system may be measured to be faster than another because its execution path length is shorter than a slower system that has additional instructions for improved error reporting and RAS. The value of the slower system, however, may be higher to the customer even though its performance is worse. And the performance of the slower system can typically be enhanced by purchasing more compute power (CPU units).

So even though a single metric may be “worse”, the customer will ultimately be looking for the system that provides the *best combination of metrics*. What are those other metrics? They are often described under the heading of “total cost of ownership.”

<sup>1</sup> CPU time is the time the Central Processing Unit needs to process the instructions in the application or benchmark.

### 6.1.1 Total cost of ownership

OS/390 environment is typically viewed as an environment that minimizes the total cost of ownership. It allows for the evolution of an organization's computing system environment, as new languages and business processes evolve.

From the perspective of this redbook, OS/390 provides a common Language Environment, that supports - in a unified way - a spectrum of languages including C, C++, Java, as well as older languages such as COBOL, Fortran, Assembler and PL/I. The Language Environment (LE) provides a *common run-time library* that is used transparently by each language to ensure consistent results across the spectrum of languages in the customer's shop.

There is a cost, in terms of extra path length, to support this commonality of results. While its path length may be slightly more than in a simple and "less evolvable" computing environment, where all of the code is in a single language, from an overall systems perspective the value of having Language Environment path length far exceeds its cost for many OS/390 customers.

One will see reports in the industry of standard benchmarks measured in simple operating environments, such as Microsoft Windows or UNIX, which show faster execution or better price/performance for a particular benchmark. Those results may be correct as a single metric, but from the customer viewpoint, they are only one of the inputs in calculating the total cost of ownership. The final cost equation must include the people, the maintainability of the system, the system run-time RAS characteristics, and the ability of the system to evolve.

Given the history of computing over the past 40 years, with new languages being developed every decade, the "value proposition" of the OS/390 Language Environment is that it provides better support for a continuing mix of new and old application functions, regardless of the language or the year in which they were written, and uniform support for common run-time library functions across all languages. Clearly, such value is recognized by the computing community at large, since organizations continue to place their critical businesses on the environment provided by OS/390.

In the long run, as the cost of raw computing continues to drop, one would expect that the value of the instructions in the "extra path length" and the value proposition of the OS/390 platform as a whole, will continue to grow.

### 6.1.2 Minimization of Language Environment path length

Given all of the above, it is nonetheless a goal of OS/390 developers to minimize the path length of customer applications, consistent with maintaining the value proposition described above.

In OS/390 V2R10, the new linkage conventions that have been described in the previous chapters provide opportunities for reducing the path length of applications written in any language that supports them. In Release 10, the C and C++ languages have been the primary focus of that support.

#### 6.1.2.1 C/C++ compiler options

The OS/390 C/C++ compiler itself provides a number of options to reduce the execution path length. Some of these options, such as OPTIMIZE, INLINE, and Inter-Procedural Analysis (IPA), reduce the amount of linkage in a program. For

information on these compiler options refer to the *OS/390 C/C++ User's Guide*, SC09-2361.

One of the purposes of this chapter is to illustrate the value of having XPLink in conjunction with other compiler options. For example, during development, one will not typically be using the compiler optimization options, because one will still be debugging. In that environment, XPLink can speed development by speeding test execution by reducing the number of instructions to be debugged. In a production environment, some customer shops will not want to use the IPA compiler optimization to simplify future problem resolution, since optimization tends to re-arrange the placement of code in the load module relative to the programmer's source description. Both development and production shops will benefit from XPLink.

#### **6.1.2.2 Compatibility mode**

In addition to supporting new or old C/C++ applications that have been completely re-compiled and re-linked, XPLink supports an environment where applications written in any language that uses non-XPLink linkage conventions still can interact transparently with XPLink applications. This is called *compatibility mode*. This mode incurs some penalty in performance, due to the extra "glue code" that must be executed in going back and forth across the different linkage stacks. These stacks are referred to as the "up-stack", "down-stack", or "no-stack" (leaf routine) and comprise the three environments across which compatibility is being maintained.

To sum up, compatibility mode protects existing application investments by providing transparent support for a *mix of applications* built with earlier linkage conventions running transparently along side new C/C++ applications built with XPLink. And its architecture allows for the evolution of the customer's computing environment into the future.

#### **6.1.2.3 The value of XPLink**

As the following sections indicate, there are real incentives for customers to recompile as much of their environment as possible, so as to exploit the reduced path length of XPLink-compiled code. And to the extent that new and old code, each built with different linkage conventions will continue to exist, this chapter gives some examples of how to evaluate overall benefit to a mixed application in a "compatibility mode" environment.

---

## **6.2 Benchmarks**

Benchmarks are of two sorts. They are either industry-level benchmarks used by multiple vendors to compare the relative aspects of their products, or development benchmarks, proprietary to each vendor. In practice, all computing system manufacturers use both sorts. In the discussion that follows, we will be using development benchmarks, a subset of industry benchmarks in a development environment, where the results of measurement are *not* used for industry-level comparisons, middleware benchmarks, or applications such as developed by solution developers.

### **6.2.1 CPU instruction-intensive development benchmarks**

CPU instruction-intensive benchmarks are C and C++ programs that exercise the instruction set of a computer central processing unit (CPU) that is configured with

essentially an ample cache and “infinite” memory and little I/O. That is, these benchmarks are not meant to be measured where a cache size or amount of memory constrains the result (a performance bottleneck). The benchmarks are used purely to evaluate the raw power of the CPU to execute the instructions generated by a given compiler/linker technology.

In this context, it is important to remember that benchmarks measure the power of the *combined* S/390 hardware and of the OS/390 Release 10 C/C++ compiler to generate sequences of instructions that lead to the best results on that hardware. In our case, we measure the benchmarks when built with and without the XPLINK compiler and the linker/binder option.

Often the same vendor will manufacture both the CPU processor and the compiler/linker pair, but there are many exceptions. There are many compilers for a given processor chip (e.g. Microsoft, Borland/Inprise, and IBM all provide compilers for the Intel chip). But it is also true, in a Linux environment, that the same compiler, gcc, generates code for multiple processors. Finally, a software emulator will allow the instruction stream of one processor to be executed on another (e.g. an S/390 emulator that executes on an Intel Pentium III processor).

Some typical development results, using a mix of CPU-intensive applications, is shown in Figure 11. Two compile-time options were used for comparison. OPT(2) is normal compiler optimization, and results in inlining some function calls so that there is no linkage code needed to get to the function. OPT(0) is no compiler optimization. As expected, XPLink improvement over non-XPLink compilation is greatest with OPT(0), where the compiler has also not attempted to optimize the program by reducing linkage. However, even with optimization in use, the effect of the improved linkage is quite significant.

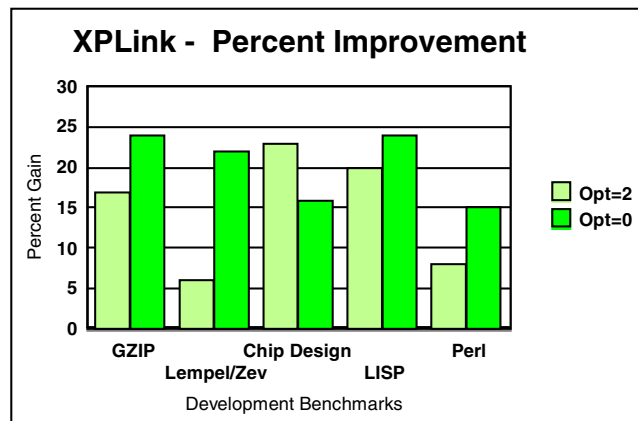


Figure 11. CPU-intensive benchmarks - percent improvement with XPLink

## 6.2.2 C run-time library

In this section we examine the performance of applications that use the C run-time library (C RTL), with and without XPLink.



### 6.2.2.1 calloc, malloc, free

The performance of the C run-time library functions for allocating and freeing memory can improve by up to 28% when used with XPLink.

In the OS/390 Release 10 UNIX environment, these functions are optimized to work with the Language Environment's Heap Pool support. (See the *OS/390 C/C++ User's Guide*, SC09-2361, and the *Language Environment for OS/390 & VM Programming Guide*, SC28-1939.)

Conversely, as shown in Figure 12, if Heap Pools are not used, the performance of these functions actually degrades, due to the cost of stack swapping. The stack swapping occurs because the Release 10 non-Heap Pool implementation still uses the "up stack" linkage convention, so that the "glue code" that interfaces XPLink code to it will create a situation in which two linkage conventions are involved. Degradation can reach 65%, as shown in the left-most column of the figure. The OS/390 Heap Pool support is really state of the art, so be sure to use it if you can.

In Figure 12, the left-hand column of each pair of results, labeled Release 9, is without XPLink; the results in the right-hand column of each pair, labeled Release 10, uses XPLink. The other abbreviations are: No HP means no Heap Pools; and HP means with Heap Pools. All results are scaled relative to the cost of doing a malloc under Release 9 (the first column value is 1.0).

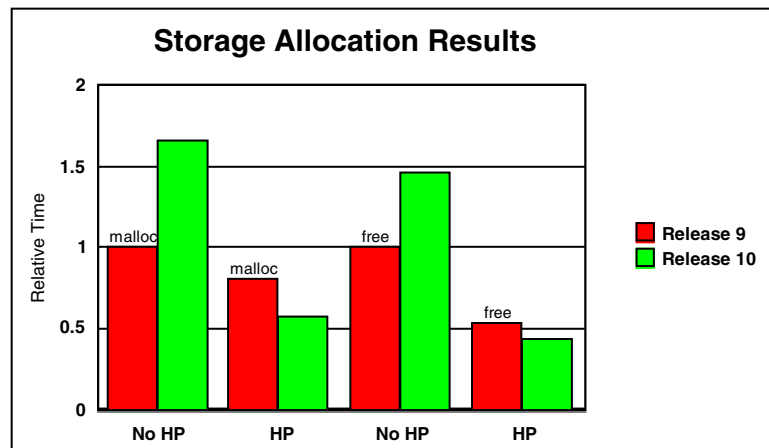


Figure 12. calloc, malloc, free - XPLink improvement and degradation

Any program with significant use of these library functions should examine the Language Environment Heap Pools storage report to profile the optimal settings for its run-time behavior, and then use those setting in the Language Environment's run-time options as described in the following steps:

1. Run your application with a representative application workload with the Heap Pools storage report turned on. This can be done by specifying:

```
export _CEE_RUNOPTS="RPTSTG(ON),HEAPPOOLS(ON)"
```

2. Rerun the application with RPTSTG(ON) and the HEAPPOOLS run-time option set to the values specified in the *Suggested Cell Sizes* from the first run.
3. The values you should specify for the HEAPPOOLS run-time option are listed in the *Suggested Percentages for current Cell Sizes* from the second run.

**Notes:**

1. The Heap Pool storage report will only show up in the storage report if HEAPPOOLS(ON) has been specified.
2. In a non-XPLink environment, to avoid a significant impact on performance, remember to turn off the storage report by specifying:

```
export _CEE_RUNOPTS="RPTSTG(OFF) "
```

3. In the XPLink environment, the RPTSTG(ON) has a minimal impact on performance.

**6.2.2.2 Pthread lock and unlock**

Thread locking and unlocking under OS/390 UNIX, as invoked from the C RTL, may show some degradation initially. This is one of the areas in which the gradual change to full XPLink implementation has not been completed within Release 10, and it is being prioritized for possible re-write in a subsequent release.

As shown in Figure 13, the extent of the degradation depends on the number of locks already on the queues at the time the lock and unlock calls are made.

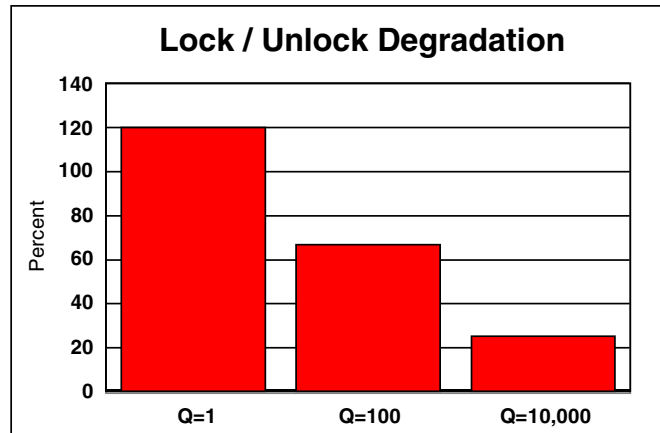


Figure 13. Pthread lock/unlock degradation with XPLink

**6.2.2.3 Math library interfaces: IEEE binary float and hex float**

The math library is an area where changes in linkage conventions can cause degradation. Users that compile their application XPLink will implicitly generate, in Release 10, a “compatibility mode” environment, if they use the hexadecimal floating point format in making calls to the C RTL.

The IBM hexadecimal floating point format C RTL Math Library has been well optimized over many decades for usage with OS Linkage conventions. The IEEE binary floating point format is a newer addition to the S/390 instruction set, and its Math Library routines are less optimal. However, its functions can be invoked from XPLink without stack swapping.

As a result, on the one hand, one will find that invoking the Hex Math Library from an XPLink application will degrade relative to invoking the same routine from the application when compiled non-XPLink. On the other hand, use of the IEEE Math

Library will be slower when invoked from a non-XPLink application, but will be somewhat faster when that application is compiled XPLink.

The following measurements show the degradation possible in the Hex Math library. (Measurements of IEEE vs. hex format are beyond the scope of this redbook.)

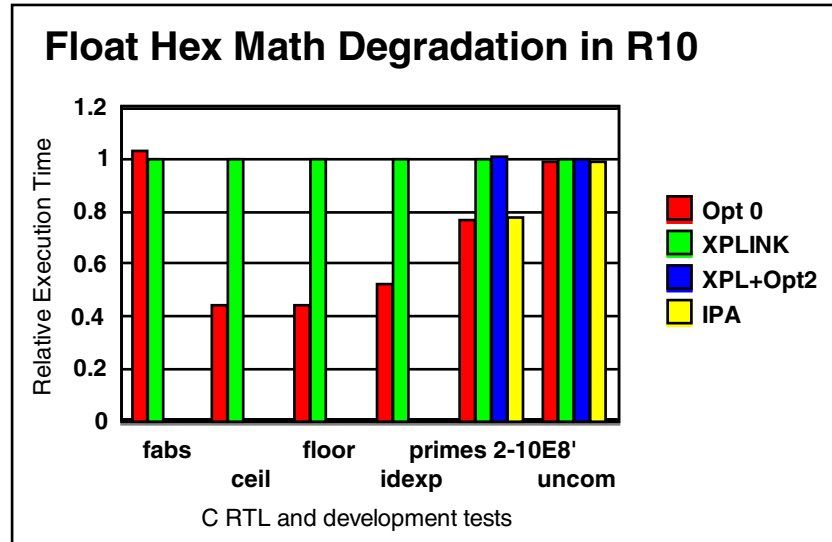


Figure 14. Potential Degradation, C RTL Hex Math Library operations

The `fabs()` C RTL call returns the absolute value of a floating number. This does not result in a call to an Assembler Language routine, and hence its time is the same from an application built with Opt 0 with or without XPLink. However, `ceil()`, `floor()`, and `idexp()` are functions from the C RTL that do make that call via OS Linkage, and their execution time per operation can more than double.

The effect of this on a simple application is shown in the “primes 2-100,000,000” test program, that counts prime numbers between 2 and 100,000,000. In this case the calls to the C RTL, while longer, are somewhat mitigated by the positive effect of the reduced linkage due to XPLink, and the overall degradation is about 30%. In this case, the user may be better off with IPA than with XPLink, at least during Release 10.

Finally, the last application is a simple “C un-comment” program, generated from lex, that has no C RTL floating point math library calls at all, and, as expected, its performance is essentially unchanged.

### 6.2.3 Industry standard compute-intensive benchmarks

Also measured were some popular technical benchmarks, which concentrate on measuring computing speed for integer and floating point arithmetic with very little I/O.

A brief description of some of these technical benchmarks and results with XPLink follows.

### 6.2.3.1 Technical benchmark descriptions

The technical benchmarks we evaluated are C versions of the following benchmarks, many of which were originally developed in Fortran:

- Linpack: A benchmark for the solution of a dense set of linear equations by Gaussian elimination.
- Whetstone: A synthetic floating-point benchmark that performs numerical calculations.
- Dhrystone: A benchmark that spends a significant amount of time on string functions. It was designed to measure the integer performance of small machines.
- Dhampstone: An adaptation of the Whetstone and Dhrystone benchmarks.
- Fibonacci: A program to test the function call overhead. It computes the Fibonacci number recursively.
- Towers of Hanoi: Another program to test the function call overhead. It solves the classic Towers of Hanoi problem.

### 6.2.3.2 Technical benchmarks

The results of running the technical benchmarks described above with the Release 10 C compiler at OPT(2) ARCH(3) with and without XPLINK are summarized in the following figures.

As expected, the benchmarks that have lots of function calls (Fibonacci and Towers of Hanoi) have the most significant improvement with XPLink, as shown in Figure 15. Fibonacci execution time is 65% of the execution time without XPLink; Towers of Hanoi is also about 25% faster.

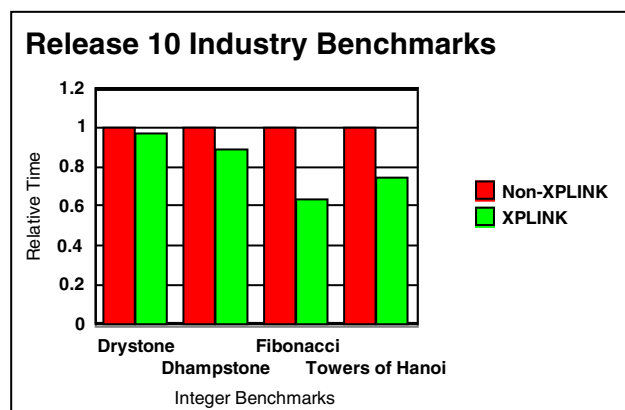


Figure 15. Industry standard integer benchmarks

Another interesting result is shown in Figure 16 on page 61. The Linpack and Whetstone benchmarks' performance degrades with XPLink when hex floating point is used, and have performance improvements when IEEE floating point is used. This is due to the former routines requiring a stack switch to access the Assembler math routines that, as of Release 10, is discussed in 6.2.2.3, "Math library interfaces: IEEE binary float and hex float" on page 58.

The results in this figure are scaled relative to each benchmark running with S/390 hexadecimal floating point, which can be significantly faster, in the Release

10 time frame, than the new IEEE format. This result is independent of whether XPLink is or is not used to build the benchmark. It is expected that in future releases, the differential between IEEE and hexadecimal floating point will lessen.

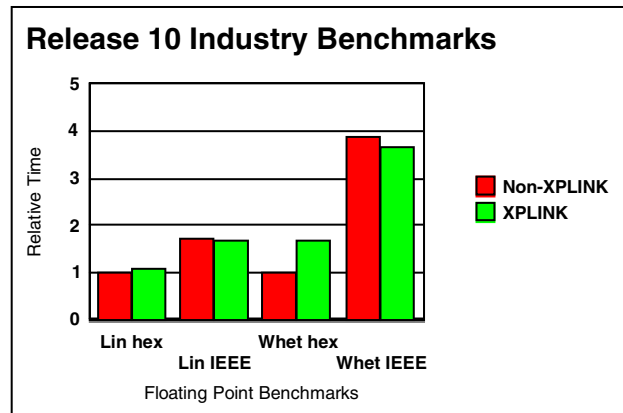


Figure 16. Industry standard floating point benchmarks

#### 6.2.4 IBM internal benchmarks

The following four benchmarks provide a range of situations in which XPLink was measured and show the expected effect.

- **Calce** is a small program that calculates the value of  $e$ ; it is very floating point-intensive, but has no program calls. No performance change is expected.
- **Pddriver** does various POSIX file system operations. No performance change is expected, since the I/O activity will overwhelm the amount of linkage code.
- **Thrasher** uses a large number of OS/390-specific UNIX System Services function calls. No performance change is expected because the code did not use portions of the C RTL that might involve stack changes.
- **Tak** is short for takeuchi, and is a highly recursive mathematical function. A significant improvement in performance was expected, due to the large amount of linkage in the recursion.

The results are shown in Figure 17 on page 62 where the non-XPLink results are normalized to 1.

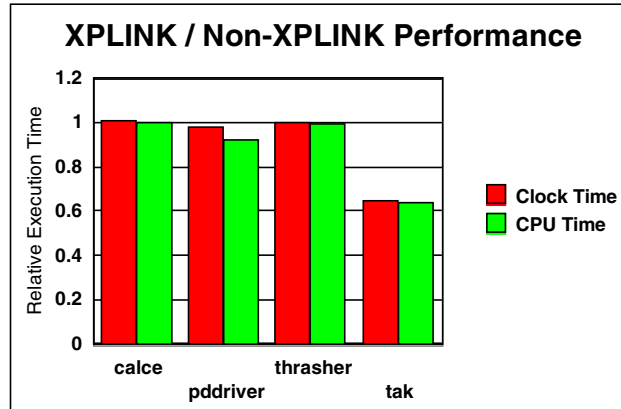


Figure 17. IBM internal benchmarks

In general, the XPLink programs were at least as fast as the non-XPLink ones, and those that used any function calls at all were speeded up slightly. For the tak program, where much of the program time is in function calls, the speedup is enormous, but for pddriver and thrasher, where most of the time spent is spent in file I/O, the speedup isn't as noticeable. For the calce program, where there are no function calls, the program runs slightly slower. This may be due to slight overhead on the beginning of an XPLink program.

## 6.2.5 Compression application

The redbook *Tuning Large C/C++ Applications On OS/390 UNIX System Services*, SG24-5606, describes a compression application that is a simple file utility that has an option to do the compression in memory rather than with real files. In memory mode, the application is greatly influenced by compiler optimization so we decided to evaluate the effect of XPLink and other Release 10 compiler enhancements on this application that is written in C and is approximately 2000 lines in two source files and two header files.

Many of the tuning actions described in *Tuning Large C/C++ Applications On OS/390 UNIX System Services* were repeated and additional tuning using XPLink and IPA Level 2 was done. The results are summarized in Table 3, which compares the effect of the compiler optimization levels on the CPU time. The measurements were done on a 9672-R26. Non-XPLink runs used the prelinker. All runs used the EDIT=NO binder option.

Table 3. Compression Application Tuning summary

Compiler options	CPU time (in seconds)
OPT(0), ARCH(3)	665.62
OPT(0), ARCH(3), INLINE	453.12
OPT(0), ARCH(3), XPLINK	507.53
OPT(2), ARCH(3), LIBANSI, INLINE	239.54
OPT(2), ARCH(3), LIBANSI, INLINE with #pragma inline	221.85
OPT(2), ARCH(3), LIBANSI, INLINE with #pragma inline, Storage using #pragma runopts	222.01

Compiler options	CPU time (in seconds)
OPT(2), ARCH(3), LIBANSI, INLINE, IPA(OBJONLY)	239.06
OPT(2), ARCH(3), LIBANSI, INLINE, IPA(LEVEL(1))	197.91
OPT(2), ARCH(3), LIBANSI, INLINE, IPA(LEVEL(2))	202.89
OPT(2), ARCH(3), LIBANSI, INLINE, XPLINK	227.94

The parameters specified on the `#pragma inline` and `#pragma runopts` directives were the same ones specified in the tuning redbook.

From Table 3, the best performance for the compression application occurred when using `IPA(LEVEL(1))`, followed by `IPA(LEVEL(2))`, `INLINE` with `#pragma inline` and then `XPLINK`. In this case, some of the additional optimizations available with `IPA` are responsible for the further improvement.

The OS/390 C/C++ Performance Team has observed some applications that perform best under `XPLink`, others under `IPA(LEVEL(1))`, and yet others under `IPA(LEVEL(2))`. The performance improvement of the application under various compiler options depends on many factors, including the number of compilation units making up the application and the number of function calls. We recommend that you try each of these three combinations on your workload to determine which combination gives the best performance for your workload.

**Note:** The performance improvements exhibited by this application may not be representative of other applications. Your applications may not respond in the same manner to compiler optimization.

### 6.2.6 The Monte Carlo application

The Monte Carlo kernel was used to calculate pi. This kernel can be used to calculate other schemes as well, any that would fit this kind of structure such as those found in High Energy Physics. This kernel calculates pi using a dartboard algorithm. It is a multi-threaded application in which each thread is totally independent of the others until the average value for pi is calculated from the values obtained by each of the threads. By *totally independent*, we mean that there are no data or variables shared between any of the threads, hence there is no mutex (to use POSIX threads terminology) to manage. Therefore, there is no contention between the participating threads. This is certainly an ideal case that programmers have always dreamed of, and which represents a good goal that one should try to reach while designing a multi-threaded application.

Given its structure, this is an application which can run as many threads as one might wish, leading to aggressive multithreading, when several hundreds, if not thousands, of threads are run concurrently.

The average gain provided by `XPLink` with this Monte Carlo kernel is in the range of 8% to 10%. (One cannot infer any valid explanation from the apparent light gain increase as the number of threads increase.) If resources are available, a gain in speedup should also be measured, as the overall MPL is increasing. So overall, for this type of aggressively multi-threaded kernel, the CPU consumption is reduced by the usage of `XPLink`, and if available resources allow, the visible speedup also benefits by the usage of `XPLink`. This latter factor should become

even more crucial as the thread awareness of the chips become more sophisticated.

The percent improvement of Monte Carlo kernel runs using XPLink versus non-XPLink with various numbers of threads is shown in Figure 18.

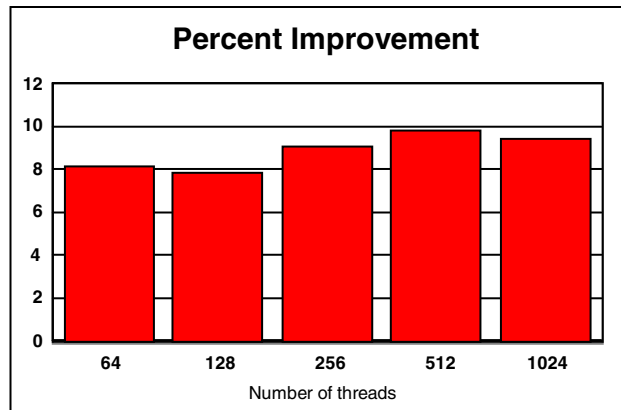


Figure 18. Monte Carlo kernel runs with various numbers of threads

---

## 6.3 Middleware benchmarks

The purpose of this section is to indicate the kinds of performance gains that middleware applications might see.

### 6.3.1 Lotus Domino

We consider the measurement of Lotus Domino to be among the most important indications of the value of XPLink because it is a complex customer environment, representative of an entire class of e-Commerce users.

#### 6.3.1.1 Domino primitives - C/C++ internal benchmarks

Lotus Domino will see performance gains when running on Release 10 due to C/C++ compiler improvements and due to XPLink. Development benchmarks of highly used functions (so-called "Domino primitives") have shown double-digit percent gains--typically a 20% reduction in path length--when measured in a development laboratory environment.

When internal development measurements of *highly used routines in the kernel* of the application indicate path length improvements of 20% to 30%, substantial improvements in both response time and throughput are anticipated. Lotus Domino may be considered typical of the kind of improvements to be seen in a large C++ application program.

#### 6.3.1.2 Mail and Calendaring Benchmark - 17.8% CPU savings

The standard measure of Lotus Notes performance is defined by a consortium of Lotus users and developers. See their URL at:

<http://www.notebench.org>

Or see:

<http://ibm.com/s390/products/domino/domperf.html>



In these measurements, the Link Pack Area (LPA) was not used because this would have required a rewrite of the PUTINLPA routine, which at this point in time is not compatible with XPLink.

### Results

The major result is that the CPU consumption was increased by 21% when not using XPLink with the R5Mail workload. *Alternatively, one can state that when using XPLink, the number of users per MIPS increased from 18 to 21.2. This is a highly significant improvement.*

Table 4 shows the environment that was measured. There were 4000 simulated users sending standard R5Mail work on an S/390 9672-XY6 LPAR, configured with 3 CPUs, 2 GB of main storage and 1 GB of expanded storage.

Without XPLink, CPU utilization was 52.2%. With XPLink, it was 42.9%. The environment for both measurements was: OS/390 V2R10; Domino R5.04a\_01; NSF buffer pool size 512 MB; DFSMS virtual size 256 MB; and DFSMS fixed size 256 MB.

Table 4. Mail and Calendaring Benchmark - 17.8% CPU savings with XPLink

	non-XPLink	XPLink
<b>Number of users</b>	4000	4000
<b>Duration of measurement</b>	1 hour	1 hour
<b>CPU utilization</b>	52.2%	42.9%
<b>DASD (SIOs/sec)</b>	1324	1285
<b>Common storage used MB</b>	1613	1636
<b>Extended storage used MB</b>	5	7
<b>Shared storage used MB</b>	1007	981
<b>EQSA size MB</b>	20	20.8
<b>ESQA size MB</b>	19.3	19.5

At this level of CPU utilization, both response times were essentially equivalent.

## 6.3.2 SAP R/3

This application shares many of the characteristics of C Language benchmarks, such as described in 6.2.3, "Industry standard compute-intensive benchmarks" on page 59. Typically, 85% of the typical SAP R/3 transaction consists of such code. One caveat to making a performance projection directly from that information would be the amount of compatibility mode linkage in the transaction (e.g. calls to libraries that have not been re-linked XPLink). To the extent that an SAP R/3 environment does not have much compatibility mode linkage, or has been completely rebuilt to use XPLink, then the gains seen for the compute-intensive benchmarks would also predict the throughput gains seen by SAP R/3.

## 6.3.3 Component Broker

We were unable to complete the IBM Internal development trace measurement in time for this publication. However, based on previous correlations of this product

with other C/C++ middleware, we would expect the improvements to be similar to those described above.

---

## 6.4 Solution developer and customer client/server applications

In this section we discuss XPLink in the context of solution developer (formerly known as Independent Software Vendor, or ISV) applications and customer client/server applications.

### 6.4.1 Measurement context

These tests have minimized the effect of client/server latency by running both the single client and the server as two address spaces in the same machine, communicating via localhost TCP/IP protocols.

The client and the server were each built with four compilation variations and were measured. As the following results indicate, the effectiveness of XPLink can be mitigated by choosing compiler options that reduce the amount of linkage in object and load modules.

#### 6.4.1.1 Compiler optimization level 0 - development

During development and in some production situations where there is a premium on being able to easily debug the applications, customers will compile their application without optimization, level 0, which is specified as OPTIMIZE(0) or NOOPTIMIZE.

In these situations, as the following results illustrate, the gain from XPLink linkage is greatest, typically 15% to 20% or more.

#### 6.4.1.2 Compiler optimization level 2 - typical optimization

The following measurements illustrate the performance gain when the application has been compiled in an optimized mode, which is specified as OPTIMIZE(2) or OPTIMIZE. Optimization reduces the advantage to be gained by XPLink, since optimization options such as inlining, which is turned on by the OPTIMIZE option, normally reduce the amount of linkage in the generated code. Many production-level applications written in C will be compiled at this level. Nonetheless, the gains here are substantial as well.

#### 6.4.1.3 Compiler InterProcedural Analysis (IPA) optimization

This option provides the best performance with high levels of compiler and binder optimization. As shown in the results below, IPA is better, typically 5% to 15% better, than XPLink alone.

In future releases, though not in Release 10, it will be possible to combine both XPLink and IPA options for even higher potential performance optimization.

### 6.4.2 net.TABLES from Data Kinetics Ltd.

The net.TABLES product from Data Kinetics Ltd. is a high performance, multiplatform table management facility for TCP/IP-enabled distributed, heterogeneous networks. It is designed for the rapid development of applications that require processing of many table lookups. net.TABLES manages tables in memory and allows multiple users across disparate platforms to read, modify and permanently store data concurrently.

Based upon years of experience with the Data Kinetics IBM mainframe product tableBASE, Data Kinetics Ltd. developed net.TABLES for system architects and application developers in the financial services, retail sales, and telecommunications industries. For full details see:

<http://tables.dkl.com/>

For more information on Net.TABLES, refer to the IBM Redbook *C/C++ Applications on OS/390 UNIX*, SG24-5992.

Three different-size environments were measured:

- 100 Tables, 10 rows per table (see Table 5 on page 68)
- 1000 Tables, 100 rows per table (see Table 6 on page 69)
- 1 Table, 1,000,000 rows per table (see Table 7 on page 70)

The benchmark results are shown in both graphical format and in tables. In the tables, the Percent Gain column (column 5) shows columns 2, 3, 4 divided by column 4, minus 1.0, and multiplied by 100.

Note that IPA usually provides the best performance, but that benchmarks built with XPLINK, OPT(2) usually provide substantial gains.

A calibration statistic, Ping Time, is shown in the last row of each table. This statistic was collected to allow additional calibration by scaling the measured results in the other rows by the relative Ping Time difference, since the measured environment was not a completely controlled one. Since the benchmark includes the entire “localhost” TCP/IP path length between the client and the server running in one machine, the measured results can be biased by any change in that environment during the measurement.

#### **6.4.2.1 100 Tables, 10 rows per table**

The first run is for a small net.TABLES environment. The first statistic, Row Insertion, is a time; the next four statistics are rates (operations per second).

In the graphs, in the ideal situation, the ratio of the Ping times should be approximately 1.0. To the extent that they are not, they indicate the TCP/IP environment was not identical *at the beginning of the measurement period*, when the Ping statistic was collected.

In general, as discussed in 6.4.2.3, “One Table, 1,000,000 rows per table” on page 70, the measurements are probably accurate to within 5%.

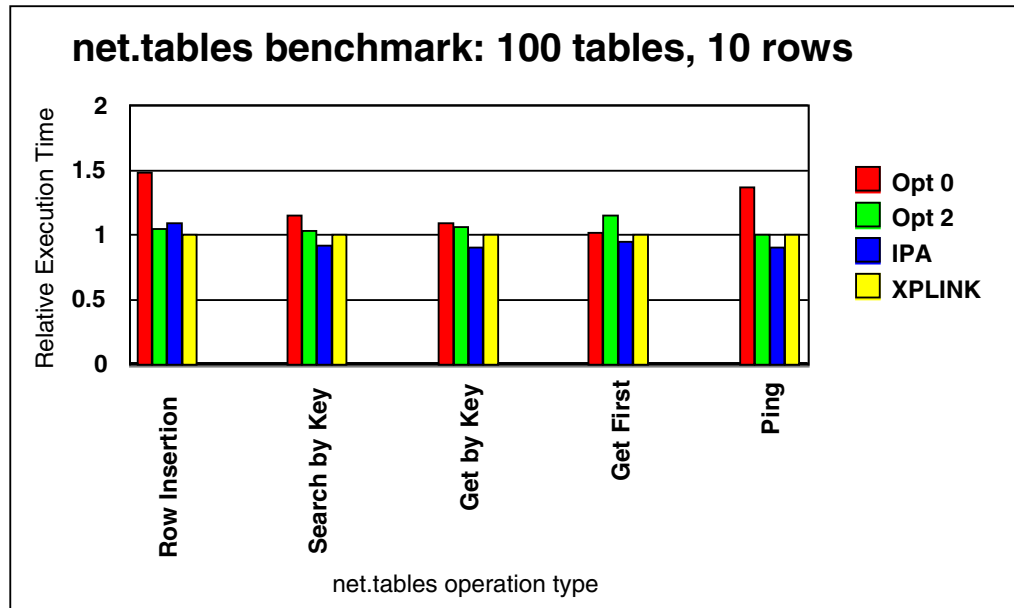


Figure 19. net.TABLES benchmark, 100 tables and 10 rows per table

Based upon the Ping statistic, the Opt(2) measurement has a ratio closest to 1.0. In that environment, XPLink shows performance gains of 2% to 14%.

Table 5. net.Tables Perf Test: 100 tables, 10 rows per table, record size = 320 bytes

Operation	Opt 0	Opt 2	IPA	XPLINK	Percent Gain
Row Insertion (seconds)	.359	.254	.262	.241	48.96% 5.21% 8.71%
Search by Key (millions/minute)	.236	.264	.294	.272	15.25% 2.85% -7.49%
Get by Key (millions/minute)	.236	.242	.283	.258	9.32% 6.38% -8/84
Get First (millions/minute)	.258	.229	.276	.263	1.93% 14.62% -4.62
Ping Time (seconds)	2.626	1.935	1.73	1.91	37.17% 1.04% -9.43%

#### 6.4.2.2 1000 Tables, 100 Rows per Table

The next set of measurements, in Table 6 on page 69, are for an environment that is considerably larger than that in Table 5 on page 68. There is a large increase in percent gain of XPLink over non-XPLink during row insertion, including IPA.

Some of this is attributable to a faster environment, as represented by the Ping Time metric.

The improvement is sometimes due to more than pure path length improvement, which at its maximum will be less than 40%. Possible explanations for an additional improvement are less page faulting or swapping due to faster overall execution. The results in Table 5 on page 68, therefore, illustrate an important point: just as a bottlenecking delay can cascade into serious degradation over other areas of an application's execution, so can a speedup, such as that due to the XPLink advantage, have a greater than nominally expected effect at the system level.

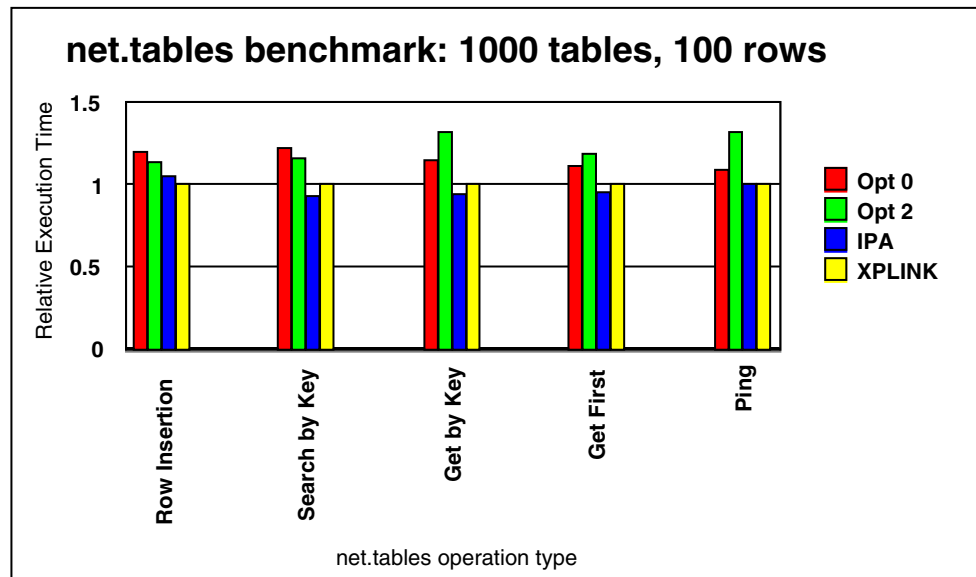


Figure 20. net.tables benchmark, 1000 tables, 100 rows per table

Based upon the Ping statistic, the Opt(0) and the IPA measurement has a ratio closest to 1.0. In that environment, XPLink shows performance gains of 15% to 20% for Opt(2), and somewhat less performance, up to 6.5% slower, than IPA.

Table 6. net.Tables Perf Test: 1000 tables, 100 rows per table, record size = 320 bytes

Operation	Opt 0	Opt 2	IPA	XPLINK	Percent Gain
Row Insertion (seconds)	38.834	36.771	34.165	32.291	20.26% 13.87% 5.80%
Search by Key (millions/minute)	.201	.212	.263	.246	22.38% 16.03% -6.47%
Get by Key (millions/minute)	.211	.183	.256	.243	15.16% 32.78% -5.08%
Get First (millions/minute)	.229	.216	.266	.256	11.79% 18.51% -3.76%

Operation	Opt 0	Opt 2	IPA	XPLINK	Percent Gain
Ping Time (seconds)	1.942	2.338	1.776	1.775	N/A

#### 6.4.2.3 One Table, 1,000,000 rows per table

Finally, measurements were made with one large table. With the exception of the Get First measurement, XPLink provided a substantial gain in performance.

The results from measuring a single large table meet expectations, except for the Get First statistic under Opt 0, which shows a higher rate than XPLink. This may be due to the nature of measuring just a single table, or it may indicate that our measurement environment is only accurate to within around 5%.

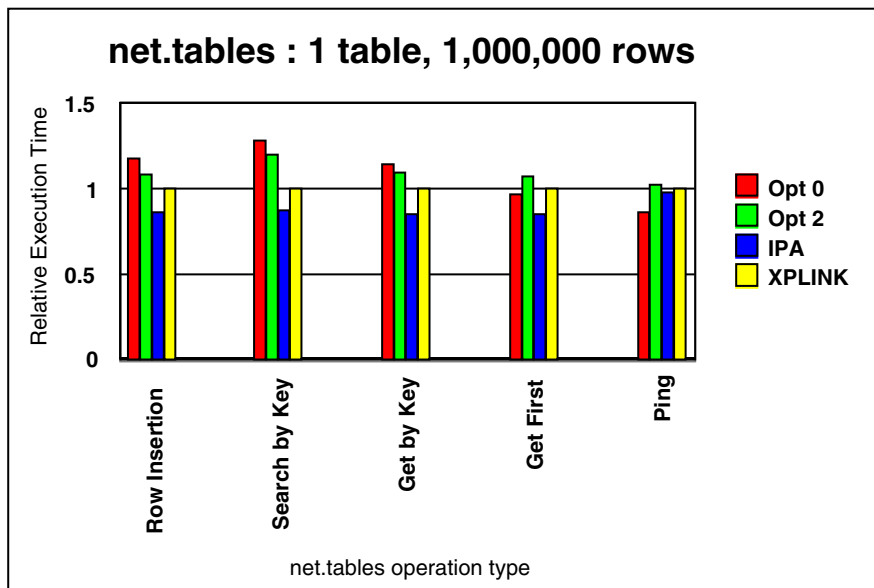


Figure 21. net.tables benchmark, one table, 1,000,000 rows

Based upon the Ping statistic, the Opt(2) measurement has a ratio closest to 1.0. In that environment, XPLink shows performance gains of 7.5% to 20%.

Table 7. net.Tables Perf Test: 1 table, 1,000,000 rows per table, record size = 320 bytes

Operation	Opt 0	Opt 2	IPA	XPLINK	Percent Gain
Row Insertion (seconds)	486.558	449.89	357.005	412.355	17.99 9.10% -13.43%
Search by Key (millions/minute)	.172	.183	.252	.220	27.90 20.21% -12.70%
Get by Key (millions/minute)	.196	.203	.264	.223	13.77% 9.85% -15.44%

Operation	Opt 0	Opt 2	IPA	XPLINK	Percent Gain
Get First (millions/minute)	.262	.234	.295	.252	-3.82% 7.69% -14.58%
Ping Time (seconds)	1.825	2.190	2.086	2.124	N/A





---

## Appendix A. XPLink code sequences

In this appendix we show annotated instruction sequences as might be generated by the C/C++ compiler in various circumstances. Circumstances other than the number of parameters or the size of the stack frame associated with the function may change the compiler's instruction-selection strategy, so these sequences should not be considered definitive.

---

### A.1 Code sequences for function prologs

The following code sequence is typical of that generated for non-XPLink function prologs. There might be minor changes to this code sequence for larger stack frames; for example, the first LA instruction becoming L/ALR, to accommodate a stack frame size larger than 12 bits.

```
* void move(int n,int currentPole,int usePole,int targetPole) {
move    DS    0D                                C entry point
        B     34(,r15)                          branch around constants
        CEE eyecatcher
        DSA size
        =A(PPA1-move)
        B     1(,r15)                            exception if entered from C++ code
        L     r15,796(,r12)                      stack extension code...
        LR     r4,r14                            ...
        BALR   r14,r15                          ...
        =F'0'                                    ...
        BR     r3                                ...not normally executed
        STM    r14,r6,12(r13)                   branch from entry point; save regs
        L      r14,76(,r13)                     end of previous stack frame (NAB)
        L      r0,8(,r15)                       add size of this stack frame
        ALR    r0,r14
        CL     r0,788(,r12)                     compare to stack floor address
        LA     r3,60(,r15)                      set up code base
        BH     20(,r15)                         go to stack extension code, if needed
        L      r15,640(,r12)                   pick up "Library Work Area" address
        STM    r15,r0,72(r14)                  save LWA and new NAB in new stack frame
        MVI    0(r14),16                       flags needed by LE
        ST     r13,4(,r14)                     save backchain
        LR     r13,r14                         move new stack frame address to GPR 13
}
```

Here is the 2-instruction code sequence generated for the same function (2K stack) using XPLink:

```
* void move(int n,int currentPole,int usePole,int targetPole) {
@1L0    DS    0D                                XPLink entrypoint marker...
        =F'12779717'                          0x0,'C',0x0,'E',0x0,'E',0x0,'1'
        =F'12910833'                          offset of PPA1
        =F'-32'                               stack frame size
        =F'2048'                               entry point
move     DS    0D                                entry point
        STM    r6,r14,8(r4)                   save registers
        AHI    r4,H'-2048'                   update stack pointer
}
```

For a stack frame in the range 2K to 4K with XPLink we get the slightly slower sequence:

```
* void move(int n,int currentPole,int usePole,int targetPole) {
@1L0    DS    0D                                XPLink entrypoint marker
        =F'12779717'                          0x0,'C',0x0,'E',0x0,'E',0x0,'1'
        =F'12910833'                          offset of PPA1
        =F'-32'                               stack frame size
        =F'4096'                               entry point
move     DS    0D                                entry point
        AHI    r4,H'-4096'                   update stack pointer
        STM    r6,r14,2056(r4)               save registers
}
```

For a stack frame in the range 4K to 32K we might get the following prolog code, the first STM instruction (STM r2,r3,...) being required only for functions expecting more than more one parameter:

```
* void move(int n,int currentPole,int usePole,int targetPole) {
@lL0      DS      0D
          =F'12779717'      XPLink entrypoint marker
          =F'12910833'      0x0,'C',0x0,'E',0x0,'E',0x0,'1'
          =F'-32'          offset of PPA1
          =F'32768'         stack frame size
move      DS      0D          entry point
          STM      r2,r3,2116(r4)  save registers containing parameters
          LR      r0,r4          keep previous stack pointer
          AHI      r4,H'-32768'    update stack pointer
          C        r4,868(,r12)    compare to stack floor address
          JL       H'85'          go to stack extension code, if needed
          STM      r5,r14,2052(r4) save registers in new stack frame
          ST       r0,2048(,r4)    backchain, possibly updated by stack extender
          LR      r2,r0          previous stack frame
          L        r2,2116(,r2)    recover 2nd argument
```

For a stack frame greater than 32K we might see:

```
* void move(int n,int currentPole,int usePole,int targetPole) {
@lL0      DS      0D
          =F'12779717'      XPLink entrypoint marker
          =F'12910833'      0x0,'C',0x0,'E',0x0,'E',0x0,'1'
          =F'-32'          offset of PPA1
          =F'32800'         stack frame size
move      DS      0D          entry point
          STM      r2,r3,2116(r4)  save registers containing parameters
          LR      r0,r4          keep previous stack pointer
          BASR     r2,0
          A        r4,148(,r2)    update stack pointer
          C        r4,868(,r12)    compare to stack floor address
          JL       H'88'          go to stack extension code, if needed
          STM      r5,r14,2052(r4) save registers in new stack frame
          ST       r0,2048(,r4)    backchain, possibly updated by stack extender
          LR      r2,r0          previous stack frame
          L        r2,2116(,r2)    recover 2nd argument
```

## Appendix B. Sample CEEDUMP header

This appendix contains the beginning of a sample CEEDUMP file. This sample was generated by having an XPLINK-compiled main call a NOXPLINK-compiled DLL function that then does a divide by zero, forcing the CEEDUMP. An analysis of the function traceback (from bottom to top) shows:

- CEEBBEXT, the bootstrap routine, is non-XPLink (with an UPSTACK DSA), which called...
- EDCZHINV, invocation of an XPLink main(), is non-XPLink (with an UPSTACK DSA), which called...
- CEEVROND, the "RunOnDownStack" glue code, is a TRANSITION DSA, which called...
- main(), is XPLink (with a DOWNSTACK DSA), which called...
- CEEVRONU, the "RunOnUpStack" glue code, is a TRANSITION DSA, which called...
- dllfunc(), is a non-XPLink DLL function (with an UPSTACK DSA). This is where the divide-by-zero exception occurred, which resulted in a call to...
- CEEHDSR, LE's non-XPLink Condition Management Dispatcher (with an UPSTACK DSA), to handle the condition.

The important thing to realize here is that the UPSTACK DSAs should be debugged using non-XPLink conventions, and the DOWNSTACK DSAs should be debugged using XPLink conventions. The TRANSITION DSAs can generally be ignored. If there is an exception in transition code, the error should be reported to IBM.

Information for enclave main

Information for thread 242A000000000000

Traceback:

DSA Addr	Program Unit	PU Addr	PU Offset	Entry	E Addr	E Offset
24017F60	CEEHDSR	10C08AF0	+000041DA	CEEHDSR	10C08AF0	+000041DA
CEEPLPKA	Call					
240174E8		23CD1400	+0000010C	dllfunc	23CD1400	+0000010C
*PATHNAM	Exception					
24017338	CEEVRONU	10CFF0A0	+00000706	CEEVRONU	10CFF0A0	+00000706
CEEPLPKA	Call					
240B76A0		23CD0388	+00000024	main	23CD0388	+00000024
*PATHNAM	Call					
240B7720		10CFDA68	+000009A4	CEEVROND	10CFDA68	+000009A4
CEEPLPKA	Call					
240170E0	EDCZHINV	23FD35F8	+0000009A	EDCZHINV	23FD35F8	+0000009A
CELHV003	Call					
24017018	CEEBBEXT	00B10860	+000001A6	CEEBBEXT	00B10860	+000001A6
CEEBINIT	Call					

Condition Information for Active Routines

Condition Information for (DSA address 240174E8)

CIB Address: 24018780

Current Condition:

CEE3209S The system detected a fixed-point divide exception (System Completion Code=0C9).

Location:

Program Unit: Entry: dllfunc Statement: Offset: +0000010C

Machine State:

ILC..... 0002 Interruption Code..... 0009

PSW..... 078D2400 A3CD150E

GPR0..... 00000000 GPR1..... 240B7EE0 GPR2..... 23CCF74C GPR3..... 23CD143A

GPR4..... 23CCF778 GPR5..... 23CCF730 GPR6..... 00000000 GPR7..... 00000000

GPR8..... 00000002 GPR9..... 10CFEA67 GPR10..... 23CD1400 GPR11..... 90CFF12A

GPR12..... 0003CAC0 GPR13..... 240174E8 GPR14..... 00000000 GPR15..... 00000005

```

Storage dump near condition, beginning at location: 23CD14FC
+000000 23CD14FC 5810D0A4 58E01000 5800D0A0 8EE00020 1DE05800 D0AC5000 C1F458D0
D00458E0 |...u.....&.A4.....|

```

Parameters, Registers, and Variables for Active Routines:

```

CEEHDSR (DSA address 24017F60):
  UPSTACK DSA
  Saved Registers:
    [snipped to save space...]
dllfunc (DSA address 240174E8):
  UPSTACK DSA
  Saved Registers:
    [snipped to save space...]
CEEVRONU (DSA address 24017338):
  TRANSITION DSA
  Saved Registers:
    [snipped to save space...]
main (DSA address 240B76A0):
  DOWNSTACK DSA
  Saved Registers:
    [snipped to save space...]
CEEVROND (DSA address 240B7720):
  TRANSITION DSA
  Saved Registers:
    [snipped to save space...]
EDCZHINV (DSA address 240170E0):
  UPSTACK DSA
  Saved Registers:
    [snipped to save space...]
CEEBBEXT (DSA address 24017018):
  UPSTACK DSA
  Saved Registers:
...

```

---

## Appendix C. Special notices

This publication is intended to help C, C++ and Assembler programmers to understand the new XPLink function in OS/390 V2R10 and later. The information in this publication is not intended as the specification of any programming interfaces that are provided by XPLink. See the PUBLICATIONS section of the IBM Programming Announcement for XPLink for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.


Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

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

IBM ®	AS/400
AT	CICS
C/370	CT
Current	DB2
Enterprise Systems Architecture/390	Language Environment

Netfinity	OS/390
PAL	RS/6000
S/390	SP
System/390	XT
400	Lotus
Lotus Notes	Domino
Notes	Redbooks
Redbooks Logo 	

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere., The Power To Manage., Anything. Anywhere., TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Københavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through The Open Group.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

---

## Appendix D. Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

---

### D.1 IBM Redbooks

For information on ordering these publications see “How to get IBM Redbooks” on page 81.

- *Tuning Large C/C++ Applications On OS/390 UNIX System Services*, SG24-5606

---

### D.2 IBM Redbooks collections

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at [ibm.com/redbooks](http://ibm.com/redbooks) for information about all the CD-ROMs offered, updates and formats.

CD-ROM Title	Collection Kit Number
IBM System/390 Redbooks Collection	SK2T-2177
IBM Networking Redbooks Collection	SK2T-6022
IBM Transaction Processing and Data Management Redbooks Collection	SK2T-8038
IBM Lotus Redbooks Collection	SK2T-8039
Tivoli Redbooks Collection	SK2T-8044
IBM AS/400 Redbooks Collection	SK2T-2849
IBM Netfinity Hardware and Software Redbooks Collection	SK2T-8046
IBM RS/6000 Redbooks Collection	SK2T-8043
IBM Application Development Redbooks Collection	SK2T-8037
IBM Enterprise Storage and Systems Management Solutions	SK3T-3694

---

### D.3 Other resources

These publications are also relevant as further information sources:

- *OS/390 V2R9.0 MVS Assembler Services Guide*, GC28-1762
- *OS/390 V2R10.0 Language Environment for OS/390 & VM Vendor Interfaces*, SY28-1152 (-08 or later)
- *OS/390 MVS Interactive Problem Control System (IPCS) User's Guide*, GC28-1756
- *OS/390 Language Environment for OS/390 & VM Debugging Guide and Run-Time Messages*, SC28-1942 (-09 or later)
- *OS/390 Language Environment Programming Guide*, SC28-1939
- *OS/390 V2R10.0 Language Environment for OS/390 Customization*, SC28-1941
- *OS/390 C/C++ User's Guide*, SC09-2361 (-06 version)
- *OS/390 DFSMS Program Management*, SC27-0806
- *OS/390 V2R10.0 UNIX System Services Command Reference*, SC28-1892
- *OS/390 V2R10.0 Language Environment for OS/390 and VM Programming Reference*, SC28-1940

- *OS/390 C/C++ Language Reference*, SC09-2360
- *OS/390 C/C++ User's Guide*, SC09-2361
- *OS/390 Language Environment for OS/390 & VM Writing Interlanguage Communication Applications*, SC28-1943 (-09 version)
- *OS/390 C/C++ Programming Guide*, SC09-2362 (-06 or later)
- *Enterprise Systems Architecture/390 Principles of Operation*, SA22-7201



---

## How to get IBM Redbooks

This section explains how both customers and IBM employees can find out about IBM Redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** [ibm.com/redbooks](http://ibm.com/redbooks)

Search for, view, download, or order hardcopy/CD-ROM Redbooks from the Redbooks Web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

Redpieces are Redbooks in progress; not all Redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**

Send orders by e-mail including information from the IBM Redbooks fax order form to:

In United States or Canada	<b>e-mail address</b> pubscan@us.ibm.com
Outside North America	Contact information is in the "How to Order" section at this site: <a href="http://www.elink.ibm.link.ibm.com/pbl/pbl">http://www.elink.ibm.link.ibm.com/pbl/pbl</a>

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	Country coordinator phone number is in the "How to Order" section at this site: <a href="http://www.elink.ibm.link.ibm.com/pbl/pbl">http://www.elink.ibm.link.ibm.com/pbl/pbl</a>

- **Fax Orders**

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	Fax phone number is in the "How to Order" section at this site: <a href="http://www.elink.ibm.link.ibm.com/pbl/pbl">http://www.elink.ibm.link.ibm.com/pbl/pbl</a>

This information was current at the time of publication, but is continually subject to change. The latest information may be found at the Redbooks Web site.

### IBM Intranet for Employees

IBM employees may register for information on workshops, residencies, and Redbooks by accessing the IBM Intranet Web site at <http://w3.itso.ibm.com/> and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may access MyNews at <http://w3.ibm.com/> for redbook, residency, and workshop announcements.

---

## IBM Redbooks fax order form

Please send me the following:

Title	Order Number	Quantity

---

First name	Last name
------------	-----------

---

Company
---------

---

Address
---------

---

City	Postal code	Country
------	-------------	---------

---

Telephone number	Telefax number	VAT number
------------------	----------------	------------

<input type="checkbox"/> Invoice to customer number	
---	--

<input type="checkbox"/> Credit card number	
---	--

---

Credit card expiration date	Card issued to	Signature
-----------------------------	----------------	-----------

**We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.**

---

## Glossary

**Address Generation Interlock (AGI)** The several cycles which must elapse between when a register is updated and when its new value is available for reference.

**Application** All the code executed from the time an executable program module is invoked until that program, and any programs it directly or indirectly calls, is terminated.

**DLL** An executable module that exports functions, variable definitions, or both, to other DLLs or DLL applications.

**DLL application** An application that references imported functions, imported variables, or both, from other DLLs.

**DLL code** Object code resulting when C or C++ source code is compiled with the DLL compiler option. C++ code and XPLINK code are always DLL code.

**Executable (program, module)** A file which can be loaded and executed on the computer. OS/390 supports two types:

**Load module** An executable residing in a PDS.

**Program object** An executable residing in a PDSE or in the HFS.

**Exported functions or variables** Functions or variables that are defined in one executable module and can be referenced from another executable module. When an exported function or variable is referenced within the executable module that defines it, the exported function or variable is also non-imported.

**Function descriptor** An internal control block containing information needed by compiled code to call a function.

**Imported functions and variables** Functions and variables that are not defined in the executable module where the reference is made, but are defined in a referenced DLL.

**Non-imported functions and variables** Functions and variables that are defined in the same executable module where a reference to them is made.

**Object (code, module)** A file output from a compiler after processing a source code module, which can subsequently be used to build an executable program module.

**Source (code, module)** A file containing a program written in a programming language.

**Variable descriptor** An internal control block containing information about the variable needed by compiled code.

**Writable Static Area (WSA)** An area of memory that is modifiable during program execution. Typically, this area contains global variables and function and variable descriptors for DLLs.

**XPLink application** An application that is made up of C and/or C++ object modules that were compiled with the XPLINK compiler option. XPLink applications are always DLL applications. Since the C/C++ run-time library for XPLink is packaged as a DLL, any XPLink executable module that calls a C/C++ run-time library is also importing from a DLL.

**XPLink code** Object code resulting when C or C++ source code is compiled with the XPLINK compiler option. XPLink code is always DLL code.



---

# Index

## Symbols

#pragma inline 63  
#pragma runopts 63  
\_\_bldxfd function 33  
\_CEE\_RUNOPTS environment variable 19

## A

AMODE 31 34  
ARCH 63  
ARCH compiler option 16  
argument passing conventions 9  
assembler code 24  
    CEEDSA macro 26  
    conventions 26  
    EDCXEPLG macro 26  
    EDCXPRLG macro 25  
    example 27  
    non-XPLink 28  
assembler language 1

## B

back chain 3  
BACKCHAIN compiler option 6  
BALR 34  
BASEREG 26  
benchmarks  
    middleware 64  
bibliography 79  
binder support 16  
binding 15

## C

C Multitasking Facility 35  
C run-time library 56  
callback function 32  
CBC.SCBCPRC 18  
CBC.SCLBDL 18  
CBCC 18  
CBCXCB 18  
CBCXCBG 18  
CEE.SCEEPROC 18  
CEE.SCEERUN 18, 23  
CEE.SCEERUN2 18  
CEEBINT 35  
CEEBXITA 35  
CEEDSA 26  
CEEEV003 23  
CEELOAD 34  
CEEPIPI 34  
CELHS001 14  
CELHS003 14  
CELHSCPP 14  
CELHV003 24  
CICS 35  
client/server applications 66

COBOL 23, 24, 34  
COMPACT 36  
compatibility considerations 23  
compiler options  
    COMPACT 36  
    DLL(CALLBACKANY) 33  
    XREF 36  
compiling 15  
CPU utilization 65  
customization 13

## D

DB2 35  
dbx 20  
DCE 35  
Debug Tool 20  
debugging 20  
DFSMS 65  
DLL style function pointer 32  
Domino primitives 64  
downward-growing stack 5  
DSASIZE 25  
Dynamic Link Library (DLL) 23  
Dynamic Storage Area (DSA) 3

## E

EDCC 18  
EDCXCB 18  
EDCXCBG 18  
EDCXEPLG 26, 28  
EDCXPRLG 25, 28  
ENTNAME 25  
executive overview 1

## F

Floating Point Register (FPR) 10  
forward chain 3  
function pointers  
    styles 32

## G

general purpose registers 3  
Generalized Object File Format (GOFF) 35  
GT2KSTK 26  
guard page 7

## H

Hex Floating Point (HFP) Math Library 24

## I

IBM C/C++ Productivity Tools 20  
IBM Redbooks 79  
IMS 35  
Initial Stack Allocation (ISA) 3, 7

INLINE compiler option 63  
installation 13  
IPA 2, 35, 36, 37, 63

## L

Language Environment 3  
    Preinitialization Services 34  
    SCEEBIND 13, 18  
    SCEECPP 13  
    SCEELIB 14  
    SCEELKEX 13  
    SCEE OBJ 13  
    SCEERUN2 14  
LIBANSI 63  
LNKLST 14  
Lotus Domino 64

## M

measurements 1  
multi-threaded applications 63  
MVS load 34

## N

NOCOMPACT 36  
NODLL style function pointer 32

## O

OS\_DOWNSTACK 31  
OS\_NOSTACK 29, 30  
OS\_UPSTACK 29, 30  
other resources 79  
overview 1  
    technical 3

## P

PARMWRDS 25  
performance  
    other initiatives 2  
Performance Analyzer 20  
performance benefits 5  
ping 70  
PL/I 24, 34  
PM3 format 17  
PSECT 26  
PUTINLPA 65

## R

REFERENCE linkage specifier 31  
register conventions  
    comparison 27  
    new 7  
    old 3  
register saving conventions 8  
resident routines 13  
restrictions 34  
RPTSTG compiler option 7, 21

runopts directive 19  
run-time library 23

## S

SCEEBIND 13, 18  
SCEECPP 13, 29  
SCEELIB 14, 29  
SCEELKED 13, 29  
SCEELKEX 13, 29  
SCEE OBJ 13, 29  
SCEERUN2 14  
SOM 35  
special notices 77  
stack frame layout 4  
stack layout 8  
stack overflow detection 7  
stack requirements  
    how to estimate 21  
STACK runtime option 21  
static stack information 6  
STEPLIB 14  
storage tuning 21

## T

team that wrote the book v  
threads 63  
tracing an XPLink application 20

## V

vendor application 66  
VM 35

## W

Writeable Static Area (WSA) 7, 24

## X

Xleaf 26  
XPLink  
    and IPA 37  
    argument passing conventions 9  
    assembler coding conventions 26  
    compatibility considerations 23  
    compiler option syntax 15  
    compiling and binding 15  
    customization 13  
    debugging 20  
    from a UNIX shell 17  
    goals 1  
    GOFF 35  
    installation 13  
    missing assembler code 24  
    performance benefits 5  
    restrictions 34  
    running an application 18  
    stack layout 8  
    storage tuning 21  
    tracing 20

- under TSO 18
  - what is new 5
  - when to convert 1
  - with JCL 18
- XPLink compiler option
  - syntax diagram 15
- XPLink compiler options
  - BACKCHAIN 6, 15
  - GUARD 16
  - NOBACKCHAIN 15
  - NOGUARD 7, 16, 21
  - NOSTOREARGS 16
  - OSCALL 16, 32
  - STOREARGS 10
- XPLINK run-time option 19
- XPLink style function pointer 32
- XREF 36





---

## IBM Redbooks review

Your feedback is valued by the Redbook authors. In particular we are interested in situations where a Redbook "made the difference" in a task or problem you encountered. Using one of the following methods, **please review the Redbook, addressing value, subject matter, structure, depth and quality as appropriate.**

- Use the online **Contact us** review redbook form found at [ibm.com/redbooks](http://ibm.com/redbooks)
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to [redbook@us.ibm.com](mailto:redbook@us.ibm.com)

<b>Document Number</b>	SG24-5991-00
<b>Redbook Title</b>	XPLink: OS/390 Extra Performance Linkage
<b>Review</b>	<div></div> <div></div> <div></div> <div></div> <div></div> <div></div>
<b>What other subjects would you like to see IBM Redbooks address?</b>	<div></div> <div></div> <div></div>
<b>Please rate your overall satisfaction:</b>	<input type="radio"/> Very Good <input type="radio"/> Good <input type="radio"/> Average <input type="radio"/> Poor
<b>Please identify yourself as belonging to one of the following groups:</b>	<input type="radio"/> Customer <input type="radio"/> Business Partner <input type="radio"/> Solution Developer <input type="radio"/> IBM, Lotus or Tivoli Employee <input type="radio"/> None of the above
<b>Your email address:</b> The data you provide here may be used to provide you with information from IBM or our business partners about our products, services or activities.	<div></div> <div><input type="radio"/> Please do not use the information collected here for future marketing or promotional contacts or other communications beyond the scope of this transaction.</div>
<b>Questions about IBM's privacy policy?</b>	The following link explains how we protect your personal information. <a href="http://ibm.com/privacy/yourprivacy/">ibm.com/privacy/yourprivacy/</a>





## XPLink: OS/390 Extra Performance Linkage

(0.2"spine)

0.17"<->0.473"

90<->249 pages







# XPLink: OS/390 Extra Performance Linkage

**New, more efficient  
function linkage**

This IBM Redbook describes XPLink, the new high performance linkage option for OS/390.

**Designed for modern  
application coding  
styles**

It discusses the means by which XPLink achieves its performance goals, and the various ways these affect the performance of C and C++ code.

**Increase performance  
up to 33%**

Finally, it shows the effect of the XPLink performance improvements on real-world applications.

## **INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION**

### **BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE**

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:**  
**[ibm.com/redbooks](http://ibm.com/redbooks)**

SG24-5991-00

ISBN 0738418471