Linkage Conventions, Module Structure and Related Matters for Little-Endian PowerPC

June 3, 1994

Rick Simpson
Peter Oden

IBM Thomas J. Watson Research Center

Last Modified - December 6, 1994

Change bars denote changes since the March 7, 1994 version.

TABLE OF CONTENTS

1 Change Highlights

December 6, 1994

- Sharpen description of parameter passing and clarify that all register parameters of a function that takes the address of *any* parameter should be stored "home" in order to materialize the entire parameter list in contiguous storage (see section 3.6.7.1-3.6.7.4 and 5.4.1-5.4.4).

June 3, 1994

- Stack frame format, parameter passing, and glue code sequences were redesigned to improve efficiency. The minimum stack frame size is now 16 bytes, and the glue code sequences no longer save and restore the link register, saving several cycles on calls to imported functions via glue. The parameter passing is changed to accommodate the new smaller stack frame. (Note: the NT implementation does not incorporate these changes. The conventions used there are now described fully in the chapter on NT-specific conventions.)
- Entry point names in the NT implementation are changed: instead of an **ep** suffix, they are identified by a "**..**" (two periods) prefix. Suffix discriminators were felt to be dangerous in the presence of the name truncation options in NT. A single period prefix is problematic because the assembler and some other utilities are not able to distinguish references to executable sections that have the same name as a user function (e.g., the entry label for a user program named data would be the same as the .data section).

March 7, 1994

- Stack frames must be a multiple of 16 bytes in size and aligned on a 16-byte boundary. This will make it possible to align quad word data items and to align data for improved cache performance.
- General register 13 is reserved for system use and should not to be accessed by compiled code. Some systems will keep a control block address in the register to avoid long paths otherwise encountered during the handling of certain interrupts.
- Parameters greater than 7 bytes long that are passed in registers must begin on an odd numbered register, and if passed in storage must be double word aligned. This helps in situations where there is no function prototype and the normal alignment requirement of a parameter cannot be inferred from an unaligned argument like a structure containing a double.
- For NT: the environment pointer is the address of the stack frame of the establisher of the environment (called the establisher frame). This is the way NT has defined it in their stack unwinding code.

- Millicode is not mapped at fixed addresses. References to it are resolved by published names. Millicode cannot be exported and so will not be called using glue sequences. The general feeling is that it is not worth reserving fixed addresses for this purpose.
- Proposal for different glue sequences to increase efficiency (to be adopted).
- Proposal for different stack frame layout to reduce minimum frame size to 16 bytes (to be adopted).
- A mechanism to check for stack overflow for frames or alloca's larger than a page-described (based on _RtlCheckStack in NT).
- Copies of function descriptors may appear in the TOC to enhance performance provided there is no name conflict with the "unique" function descriptor for a function.
- Some refinements to the description of stack unwinding when encountering millicode. These include definition of some codes appearing in the HandlerData field of the function table to allow identification of the type of the millicaode.

September 24, 1993

- Assembler syntax [tocv]x and .section x,"t" defined for data in TOC. Symbol attribute "TOC" and RLD flag "DATAINTOC" defined.
- Platform specific definitions segregated from general ones.
- For NT, entry point names changed to e.g., foo.ep (from .foo) to avoid conflict with object and executable section names such as .data.

July 25, 1993

- Prologue code restrictions needed to permit stack unwinding spelled out in detail and guidelines given for prologue code formation. Ditto for epilogue code.
- Data in TOC is permitted. Additional assembler syntax to be defined. New RLD type SSREL (subsection relative) and subsection .idata$t defined to help support this.
- Names of entry points for register save/restore millicode now begin with "._" to reduce the chances of name conflicts.
- Slack space at end of stack frame defined as 232 bytes. This is enough for a full register save, including link, TOC, and condition registers. More is not warranted for NT due to small kernel stack sizes.
- Locations for a procedure to save its link, TOC, and condition registers are in its own stack frame. There is no need to specify special locations for these. For NT, reverse execution of the prologueprovides correct stack unwinding. Tag table entries specifying save locations can be defined for other systems when needed.
- Two reserved locations for saving registers by glue code defined in stack frame header. An additional word is reserved for future use. Two others are retained as "spares" for special purpose utilities such as profilers.
- Reverse execution of prologue code mechanism described.

June 25, 1993

- Default compiler linkage is local (to a function in the same executable). Investigation shows that this is the predominant mode and should therefore be made the default.

- "Automatic" insertion of *glue* when local call turns out to be imported. Slots in header reserved for use by *glue* code when saving rtoc and the link register. This permits linking to code in libraries even if no special pragmas have been specified.
- Tag table index and tag table respecified for NT port to accord with *function table* and *scope table*. Restriction on prologues for exception processing: they must be able to be "executed backwards" by system stack unwinder to restore saved registers.
- For NT port, TOC will be mapped into the *.idata* section as part of the *import address table*.
- Naming convention change: for program foo, the name of its code section is now .foo (rather than @foo). The "@" symbol has a special meaning to the NT linker.
- Stack frame header has architected save locations only for the backchain and for use by *glue* code. The remaining 3 fullwords are spares for use by utilities such as profilers. No other save locations need be specially reserved.
- The section on the PE module format is being revised. The discussion on exception handling is being rewritten to include more specific information on the implementation for the NT port.

2 Introduction

This document describes conventions adopted for Little-Endian PowerPC. It is intended to aid those developing compilers and those writing low-level assembler language code for operating systems on that architecture. Although the conventions are derived from those for AIX, little familiarity with those conventions is assumed. Assembler language examples are given to show the sort of code that compilers should generate for specific situations. A detailed definition of PowerPC assembler language is *not* included here -- see the relevant assembler language reference manuals.

The conventions related to the Table of Contents (TOC) are adapted from those of AIX on RISC System/6000 to the exigencies of the PowerPC architecture. It is intended that there not be gratuitous differences.

Unless otherwise noted, the conventions and examples in this document are intended to apply to all systems developed for little-endian mode operation on PowerPC architectures. System-specific sections detail aspects that are particular to Windows NT and WorkPlace.

2.1 Motivation for changes

Existing conventions from AIX on Power have been adopted wherever appropriate. Changes have been made in some areas. Primary reasons for change include:

- **Hardware differences.** The most pervasive difference is that alignment of data is an exemely important factor affecting performance. The penalty for storage access to unaligned data is so severe that every effort should be made to align data properly, and in cases where that is not possible, compilers should generate in line sequences to handle unaligned data.

  Another important difference is the lack of load multiple word and store multiple word instructions in Little-Endian PowerPC. This affects the way register saving and restoring is handled at function entry and exit.

  Other hardware differences include lack of string instructions on Little-Endian PowerPC, the existence in the architecture of "optional" instructions, and architecturally required instructions with implementations that differ between the 601 processor and later processors. These differences drove the definition of "millicode" in Section 3.11.

- **Compiler optimization of calling sequences.** As in AIX, all calls to functions are generated as if the call target were bound with the caller into a single executable. Calls outside the executable (to shared libraries) are discovered to be different at link-edit time, and the linker inserts "glue" code to save the caller's TOC pointer, load the target's TOC pointer and entry address, and branch to the target.

  The "glue" code is as efficient as it can be, but it cannot be scheduled by the compiler because it is spliced in after compilation. There is no opportunity to interleave the "glue" instructions with other, unrelated instructions in the compiled code, such as the instructions that load the parameters being passed. There is no chance for the "glue" code to re-use values in registers — two calls in a row to the same target cannot take advantage of the fact that the first call has loaded both the target's TOC address and entry address and avoid loading them again. These optimizations are possible if the compiled code itself performs the "glue" function.

For this reason, compiler options and #pragmas are introduced for the the calling convention on Little-Endian PowerPC that allow specifying at compile time that certain calls are to routines outside the bound module (see Section 3.6.6). When so specified, the instructions that save and restore the caller's TOC register, load the target's TOC pointer, develop the target address and branch to it are generated "in-line" by the compiler; no "glue" is added by the linker. These instructions are subject to all the compiler's optimizations.

- **Performance and space enhancements.** In order to streamline linkage to imported functions, function descriptors consist of two rather than three words. For C programs, only the first two words of the AIX function descriptor were used anyway, and a mechanism using only two-word function descriptors was designed to do "up level" addressing for nested functions that occur in languages like Pascal and PL/1. That eliminates an instruction in the sequence used to implement calls via function pointers as well as saving a word of storage.

  A slight change in the way arguments are passed provides another improvement. Each stack frame for a non-leaf procedure in AIX contains eight words corresponding to the eight parameter general purpose registers. This storage in a caller is only used by callees that take the address of their parameters. A different mechanism is incorporated in these conventions to achieve exactly the same function but without requiring every non-leaf function to reserve eight words. Coupled with a reduction in the size of the stack frame header to four words (from six), this results in a minimum stack frame size of only 16 bytes--an important potential saving for highly recursive programs running in small memories.

- **Descriptive information about procedures.** Tables that describe information needed for traceback, debugging, and stack unwinding are no longer embedded in the code as is done in AIX. This achieves better performance through more efficient cache utilization. In addition, information is included to support language specific frame-based exception processing such as Microsoft's structured exception handling and "throw-catch" in C++.

- **Lack of underlying support for certain AIX concepts.** The primary example here is AIX's csects, which have both name and storage class and are the replaceable units that the AIX linker deals with. This concept is missing from Little-Endian PowerPC operating systems. The MIPS port of Windows NT, which the Little-Endian PowerPC port is expected to resemble closely, has only the original Unix and COFF concept of "sections" within which labels are defined. This led to changing the assembly language definition from AIX, dropping the .csect operator and substituting .text, .data, and the like.

Since Little-Endian PowerPC cannot execute binaries compiled for AIX/Power or even AIX/PowerPC (Big-Endian), we are not constrained by the need to maintain binary compatibility with AIX.

2.2 Intent

This document lays out the conventions to be followed by compilers, authors of assembly routines, and linkers so as to enable communication of the necessary information among these functions to permit production of executable files.

The authors hold that the linker and loader should depend only upon the minimum amount of information necessary to perform their functions. Introducing extraneous externally visible names, for example, whether functionally derived from other names or not, should be avoided as much as possible. In addition to being a gratuitously uneconomical representation, such a practice increases the risk of name clashes,

However, this minimalist approach ought not be construed so as to preclude embedding useful additional information in object files for use by programs such as debuggers and profilers, or for post compile-time processing to improve performance. Of course, care should be taken to prevent standard linkers and loaders from developing a dependency on such information; and its presence should not in any way compromise their normal functioning.

3 Linkage Conventions — General

3.1 Table of Contents (TOC)

The PowerPC instruction set, like that of POWER (RS/6000) before it, lends itself naturally to compilation of position-independent code. In fact, it is difficult to write position ***de***pendent code because PowerPC has no instruction formats that support the embedding of a memory address in an instruction[1]. Instead, instructions that reference memory involve a base address in a general register and optionally a displacement field in the instruction (signed 16-bit constant) or an index value in a second general register.

Given that an address in a register must be used in order to load anything from storage, a software convention is used to make it easy to load the addresses of external variables, functions to be called, and the like. The addresses of the set of items within an executable that must be so addressable are collected by the linker into a vector called the Table of Contents, or TOC. There is one TOC per executable; all functions within it use the same table for loading pointers to external variables or routines. Consider an executable made up of routines a() and b(), and external variables x and y. These may be compiled separately or may all be in one source file. When these four items are linked to form an executable, the TOC structure will be as shown in Figure 1.

Within the TOC, each distinct address occurs in only one entry and this entry is used by all procedures that access the data at that address. In general, the displacements of entries in the TOC cannot be known before the linker executes. Language processors generate relocation information to enable the linker to adjust the displacement fields of instructions that reference items in the TOC.

Separating the addressability of a function's data from its code address is one of the salient advantages of the TOC scheme. It permits sharing code among several separate programs (or threads that don't share static data) running in the same address space. It achieves this without requiring the pre-assignment of code addresses for all the functions involved in such an environment. This feature assumes increasing importance as network distribution of executing code increases.

3.1.1 TOC pointer register

By convention, the base of addressing for a procedure's environment (generally the address of its TOC) is passed to a called function in general register 2. The called routine must treat this value as non-volatile, ensuring that the same value is in register 2 after returning. However, there will be times when another value must be placed in register 2, for example when the routine sets up a call to another that requires a different base address. In that case, the value that was in register 2 on input to a procedure must be saved so that, on normal[2] exit from the procedure, the value will be restored to register 2 by epilogue code.

1. The "absolute" variant of the unconditional branch instruction is the only exception, and the address it supports is limited to 26 bits. While it would be possible to load a 32-bit address from the instruction stream using two successive instructions, each with a 16-bit immediate value, this approach becomes impossibly inefficient for 64-bit addresses.

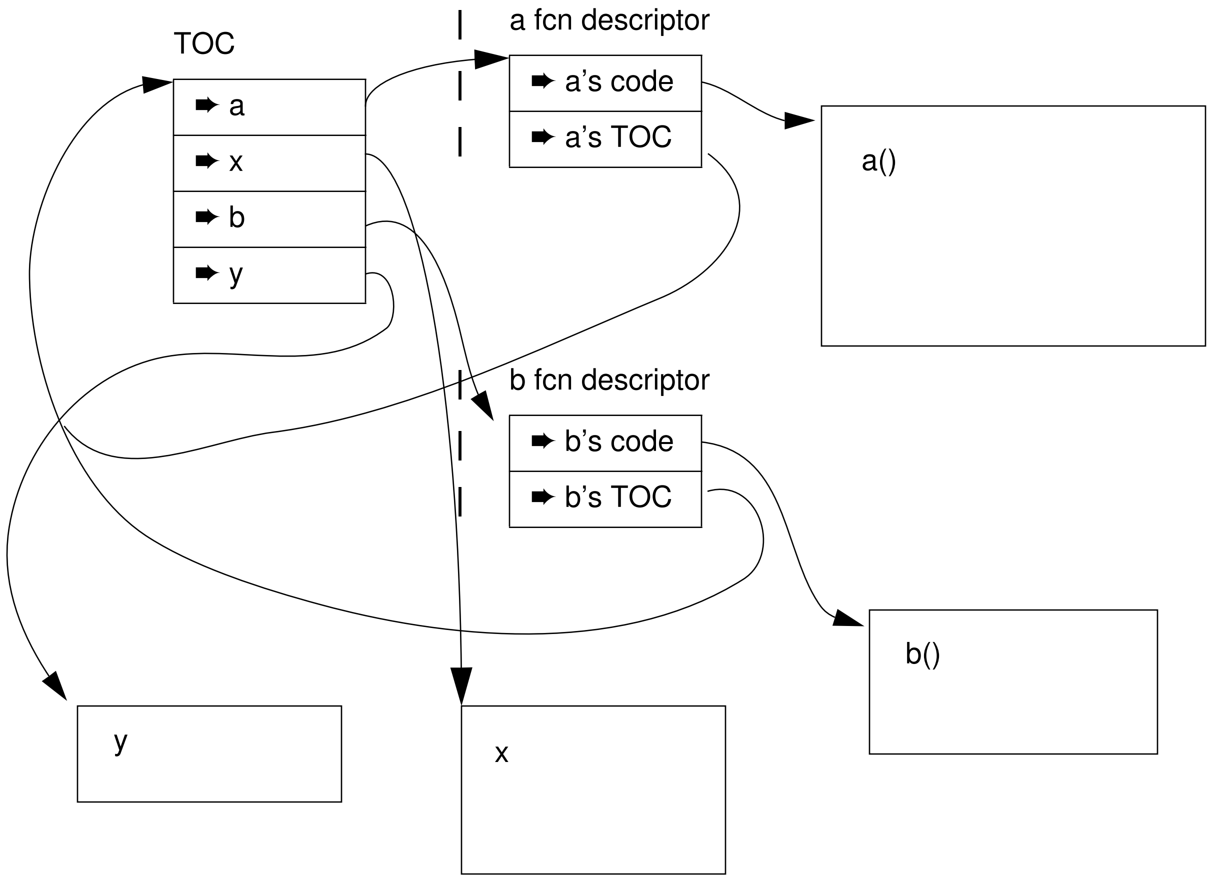

Figure 1 Addressability to externals via the TOC

3.1.2 Function descriptors

When one routine calls another, the caller must pass the address of the called routine's TOC in register 2. Unless the called routine shares a TOC with its caller, code must be produced to get the new TOC address into register 2. The required value can be found in the called procedure's *function descriptor*, a two-word data item that contains:

- The address of the called routine's code (the entry point address), and
- The address of the TOC for the called routine, which may or may not be the same as the TOC address for the calling routine[1].

As is the case for all data addressed via the TOC, the address of the function descriptor will be found in the caller's TOC. The name of a function (such as a and b in Figure 1) is associated with the address of the function descriptor, not the address of the code. It is the address of the function descriptor that is used when assigning to a function pointer or passing a pointer to a func-

2. Abnormal exits are exceptional conditions. The value in register 2 on entry must be saved in such a way that the exception handlers know how to restore register 2 during such processes as "stack unwinding".This point is discussed further in the section on exception handling.
1. For languages that allow taking the address of a nested procedure such as Pascal, the second word of the function descriptor for such a procedure is the address of its environment (i.e., a pointer to the stack frame of its statically containing procedure). The address of the TOC is a part of the environment of a nested procedure and is therefore computable from its environment address.

tion. In this way a single pointer suffices to identify both the instructions and data (TOC address) for a routine to be called.

Each function (in the read-only .text section) has exactly one function descriptor (in the process' read-write .data section). The compiler generates the function descriptor at the same time it generates the executable code. Two function pointers are considered to compare equal if they point to the same function, which will be the case precisely when they have the same value: the address of the unique function descriptor.

Although multiple function descriptors that point to the same function cannot in general be allowed because of the semantics of comparison of function pointers, it is sometimes useful to have the function descriptor or a copy of it residing in a TOC. The unique (named) function descriptor can be treated like any other data item and can be mapped into the TOC associated with the function's code. When making a copy in another TOC, care must be taken not to expose its name or use *its* address when comparing function pointers. Implementing such a feature requires linker support, but may increase the efficiency of calls to the function since it eliminates one addressing indirection. Such copies of function descriptors are sometimes referred to as *local function descriptors*.

3.1.3 Assembler Notation for TOC entries

Although not all entries in the TOC have externally visible names, some way of denoting them in assembly code is required, and it is useful to be able to refer to them in the examples included in this document. The notation

```
[toc]bletch
```

stands for the entry in the TOC that, at execution time, contains the address of the data named "bletch". Equivalently, it is used to represent the *displacement* of that entry from the origin of the TOC. This notation is not and does not of itself give rise to a global symbol.

3.1.4 Assembler Notation for data in the TOC

The original TOC contained only address constants. It is, of course, possible to add data items to the TOC. To specify that a data item, say an integer named gorp, is expected to be mapped into the TOC, the assembler syntax is

```
.toc
gorp:    .long
```

Whether or not a function contains the defining instance, the symbol would be addressed using the TOC register as a base, and the notation

```
[tocv]gorp
```

stands for the displacement of the entry in the TOC that, at execution time, contains the data item named "gorp". The symbol gorp is associated with the *absolute address* of that entry rather than with its displacement from the origin of the TOC. Note that neither of these notations denotes [toc]gorp, which may also be present in a program. As before, [tocv]gorp is not and does not of itself give rise to a global symbol.

3.1.5 Other assembler notations

These follow the general conventions for most assemblers. For details, see the assembler reference manuals.

3.2 Register Usage

3.2.1 General purpose registers

General purpose registers are classified into four (not necessarily disjoint) sets:

1 Reserved registers are not available for use by compilers. Values in them and how they are used are system specific. Only one register is reserved.

gpr13	Not to be used by compilers.

2 Dedicated general purpose registers are reserved for special uses. They define specific values that are expected to be available either at all times or at certain times, such as when a function is called.

gpr1	Stack frame pointer, always valid
gpr2	TOC (or env.) pointer on entry to, and return from, a procedure
gpr3 – 10	Used to pass parameters and return values

3 Non-volatile general purpose registers may be presumed to have the same values following a procedure call as before the procedure call.

gpr1	(must always point to current top of stack)
gpr2	
gpr14 – 31	

4 Volatile general purpose registers may *not* be presumed to have the same values frollowing a procedure call as before the procedure call.

gpr0, 3 – 12	volatile

3.2.2 Floating point registers

Apart from those that are used to pass and return floating values to functions, floating point registers have no dedicated uses.

fpr0	volatile
fpr1 – 13	volatile and used for passing parameters and return values
fpr14 – 31	non-volatile

3.2.3 Condition register

The 32-bit Condition Register (CR) is divided into eight 4-bit fields, numbered 0 through 7. Some PowerPC instructions deal with individual bits in the CR, some deal with 4-bit fields, and some deal with the entire 32-bit register. See *PowerPC User Instruction Set Architecture* (Book I) for details.

The linkage conventions treat the CR as a set of eight 4-bit fields. Each field can hold the result of, for example, a comparison. Certain of the fields are volatile across calls (value in the field need not be preserved), and others are non-volatile (value on entry must be returned on exit).

cr0 – 1	volatile
cr2 – 4	non-volatile

cr5 – 7 volatile[1]

The move from condition register (mfcr) instruction can copy the entire CR to a general register. The move to condition register fields (mtcrf) instruction can copy one or more fields from a general register to the CR.

3.2.4 Other registers

Certain other registers are accessible to user code and must be dealt with as part of the calling convention.

LR	Link Register	volatile
CTR	Count Register	volatile
XER	Fixed Point Exception Register	volatile
FPSCR	Floating Point Status and Control Register	volatile

The values in these registers after a call cannot be presumed to be the same as before the call. Thus, there is no need for a function that uses such a register to preserve its original value.

3.3 Stack Frame Layout

Stack frames are allocated from high address to low address. General register 1 is presumed always to point to a valid stack frame, and the contents of the fullword at that address always points to the previously allocated stack frame. One[2] read-only page, or unmapped page, is associated with addresses at the (low address) end of the stack to act as a guard page. This permits a single instruction (stwu) to be used to acquire a stack frame and simultaneously check for stack overflow for frames not exceeding a page in size.

Stack frames must be aligned on 16-byte boundaries. That is, they must begin on a 16-byte boundary and their lengths must be a multiple of 16 bytes.

Figure 2 shows a diagram of the stack frame[3].

3.3.1 Stack frame header

3.3.1.1 Header format

The stack frame header contains four 32-bit words:

Offset (hex)	Contents
0	Back chain; points to caller’s (next higher addressed) stack frame
4	Reserved for use by called function (e.g., to save link register)
8	Slot for glue code to save a register
C	(spare)

1. This is a slight change from the original AIX convention, in which cr.5 was originally reserved for "global" use. AIX has since changed cr.5 to "non-volatile" and finally to "volatile".
2. A system may decide to use more than one page. The use of stwu to check for stack overflow works correctly for sizes up to the number of reserved pages (up to eight pages).
3. The stack frame layout implemented for the NT port is different and is described in section 5.2

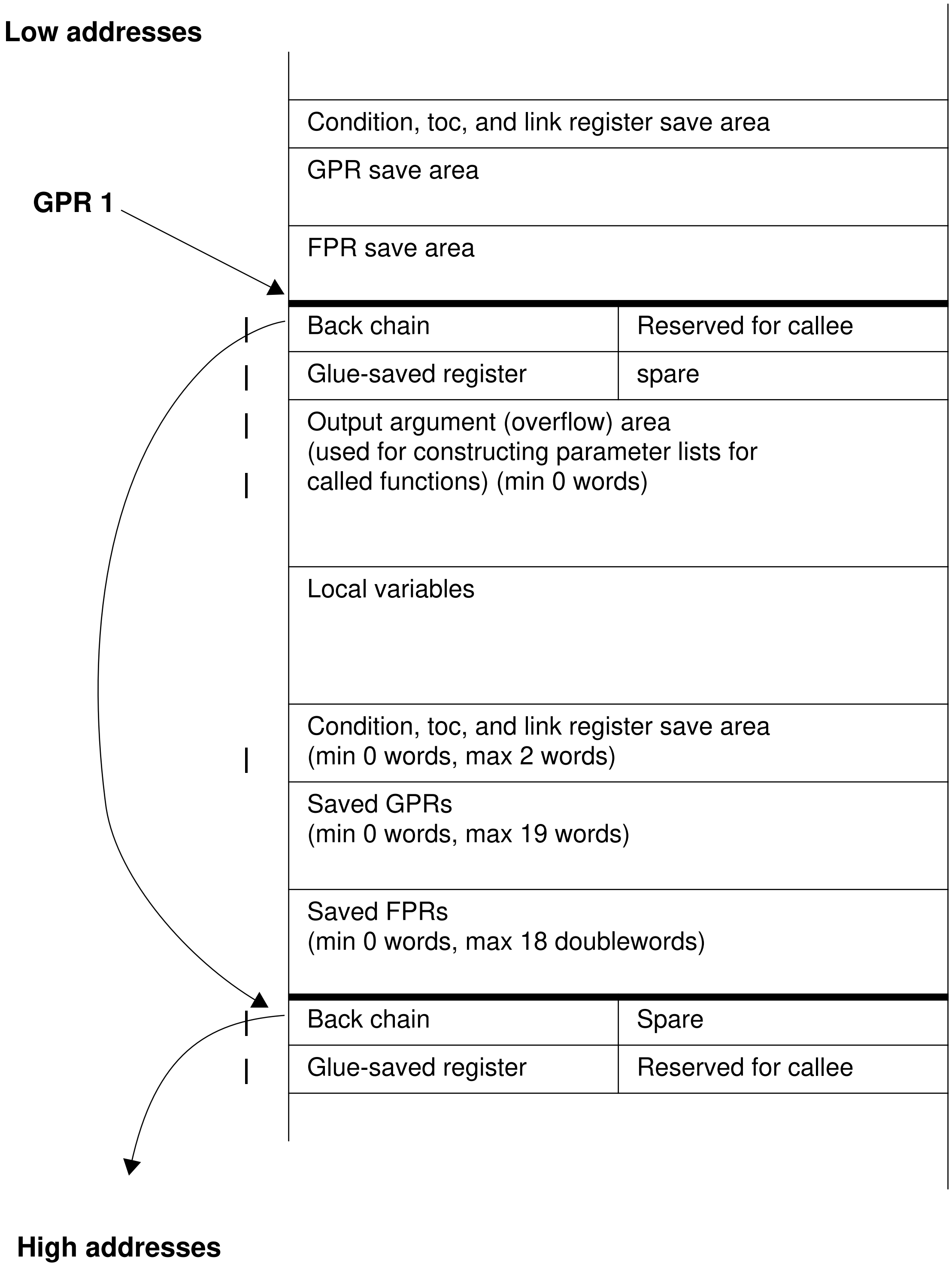

Figure 2 Stack frame format

3.3.1.2 "Ownership" of stack frame header

Except for the back chain, a function *never* stores into any other words of its "own" stack frame header. Thus while a function is actively executing, only the back chain slot in its stack frame header contains useful data. This allows an efficient implementation of the alloca() function, which need only move the back chain field to a lower address in order quickly to allocate additional storage in the stack frame. Linker-inserted "glue" code (see Section 3.6.4) may use a reserved location in the header to save a register, usually register 2. One location is reserved for use by the called function. This allows, for example, a highly recursive function to save the link register in its caller's stack frame header, achieving a minimum stack frame size of 16 bytes. The remaining word (at offset 4) is a "spare" that may be used by specialized utilities such as profilers, tracers, debuggers, etc..

3.3.2 Output argument area

This portion of the stack frame is used when passing more arguments to a called function than will fit into the available parameter registers. In most cases, it will not appear at all. See Section 3.6.7, "Parameter passing" on page 25, for details.

3.3.3 Local variables

This is the dynamic storage area for the function and can be any size. It contains such items as "automatic" variables, temporaries and register spills used by the compiled code.

3.3.4 Saved floating point registers

If the function uses any floating point registers from the "non-volatile" set, their values on input to the function are saved here. For the NT port, the code to save the registers must be a part of the procedure's prologue; see Section 3.6.8, "Register saving/restoring" on page 32. No space need be allocated for this area if no floating point registers are saved.

3.3.5 Saved general purpose registers

If the function uses any general registers from the "non-volatile" set, their values on input to the function are saved here; see Section 3.6.8, "Register saving/restoring" on page 32. No space need be allocated for this area if no general registers are saved.

3.3.6 Saved special purpose registers

Whenever a function changes the values in the link register, r.toc (gpr 2), or the non-volatile portion of the condition register during the course of its execution, it must restore the changed registers prior to exiting; see Section 3.6.8, "Register saving/restoring" on page 32. Although one word is always available for use by the callee in the caller's stack frame header, additional space in the callee's stack frame can be allocated as necessary.

3.3.7 Slack space at end of stack

During function prologues, register values may be saved on the stack at negative offsets of up to 232 bytes from the stack pointer prior to the acquisition of the new stack frame. That is sufficient space to allow a full save of all non-volatile registers plus the link, TOC, and condition registers. This means that at some times data may reside in locations beyond the current "top of stack", defined by the value in r.sp. To protect such data, routines that may gain control at arbitrary times (such as exception or interrupt handlers running on the same stack) should avoid using the slack space: addresses lying within -232 bytes of the current stack pointer.

This slack space is useful not only for saving registers prior to resetting the stack pointer, but also because calls to leaf procedures or procedures that optimizing compilers can treat as leaf procedures may avoid acquiring a stack frame altogether.

3.3.8 Minimum stack frame size

A program need not acquire a stack frame at all if it

- requires no stack storage outside the "slack space", taking account of any other routines that may be using the slack space simultaneously, **and**
- calls no other functions that use the stack.

If a stack frame must be obtained, the minimum size frame consists of the header (4 words = 16 bytes).

3.4 Checking for stack overflow

The store-with-update (stwu) instruction is used to decrement the stack pointer by the length of the new stack frame and to store the back chain (address of previous stack frame) all in one operation. For moderate-size stack frames this store instruction suffices to check for stack overflow because the "guard page" is either unmapped or is mapped read-only at the limit of the stack, thus causing the stwu to fail when the stack limit is exceeded. For stack frames that are larger than the a page, decrementing the stack pointer may step it over the guard page and thus skip the stack overflow test. When such large stack frames are generated, language processors should insert code in the prologue to test for stack overflow explicitly. A typical sequence is

```
mflr   r.0                 # prepare to save link register
li     r.12,-frame_size    # put negative of frame size in r.12
stw    r.0,4(r.1)          # save link register
bl     _RtlCheckStack[1]   # run-time routine causes interrupt if overflow would occur
stwux  r.1,r.12(r.1)       # acquire the frame and store the back chain
```

Of course, if the desired frame size is only a few pages in size, a compiler may choose to generate more efficient code than branching to a checking routine; for example, in line code that "touches" each page up to the desired size.

3.5 Naming Conventions

3.5.1 Function names

Since a function descriptor is associated with every function, two external names are needed. For a function named "foo" they are:

- The name of the first instruction of the code (the actual entry point to which one can branch). This will be "**.foo**" in our example[2].

1. For purposes of this illustration, this is presumed to be a special run time routine that preserves all registers except r.0 and r.11; and expects the negative of the desired frame size in r.12.
2. Prefixing the function name with "." is the general convention adopted in this document both in the text and in the examples.. Specific platforms may require other conventions to avoid name conflicts.

- The name of the 2-word function descriptor containing the pointer to the function's code and the pointer to its table-of-contents (TOC). This will be "**foo**" in our example (that is, the unadorned name).

The unadorned name is the one that is used to reference a function — the address it denotes would be the value assigned to a function pointer, for example.

Except when writing assembler code, programmers generally need not be aware of the entry name for a function. The only name used in source code and imports and exports lists is the unadorned function name. The entry name is automatically generated by the compiler, together with the appropriate RLDs and symbol table entries to allow the linker to make the right connections. Entry names also appear in special stub sequences called glue code used in implementing calls to functions in shared libraries. These sequences are automatically introduced by the linker. Source language debuggers should allow users to use function names when setting entry point break points.

3.5.2 Variable names

External variables are treated similarly to functions, except that there is no need for the equivalent of a "function descriptor". For an external variable named "bar", we just use the unadorned name "**bar**".

Figure 3 shows how the various names and pointers are related to one another.

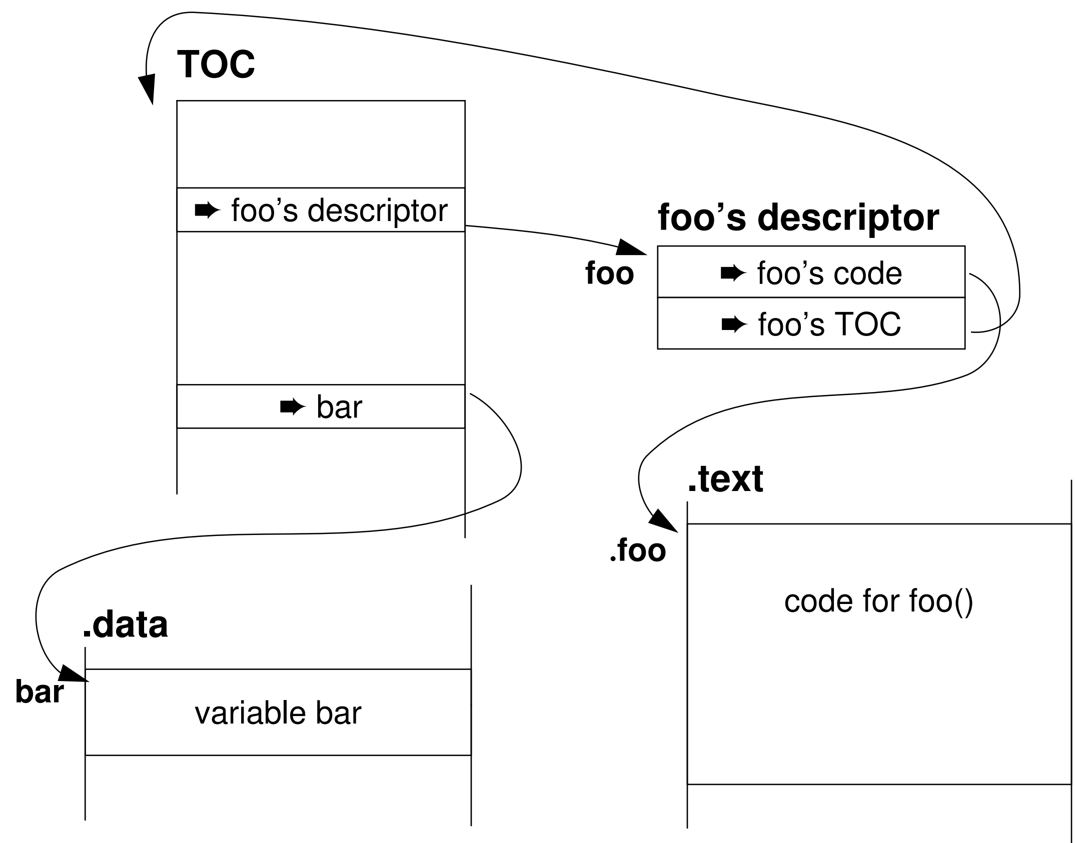

Figure 3 Showing how the TOC, function descriptors, functions, and data are connected.

3.5.3 Entry point of executable module

An executable module has a single entry point, whose value is stored in the module header. For Little-Endian PowerPC, this will be the **address of the function descriptor for the first function to be executed.** From the descriptor can be found both the address of the first instruction and the address that must be loaded into the TOC register.

3.6 Calling Sequence

3.6.1 Default local linkage straight to code

Unless otherwise specified, compilers can generate direct calls to functions using simply a relative branch and link instruction. This will result in very efficient linkage when

- The target and the caller are bound in the same executable, i.e., share a TOC, and
- The target can be reached via a relative branch computed by the linker (no relocation needed at run time).

Figure 4 shows how linkage works in this case. The "lwz" instruction shown in brackets indicates that the caller's TOC address is moved into r.toc (general register 2). Of course, this instruction is

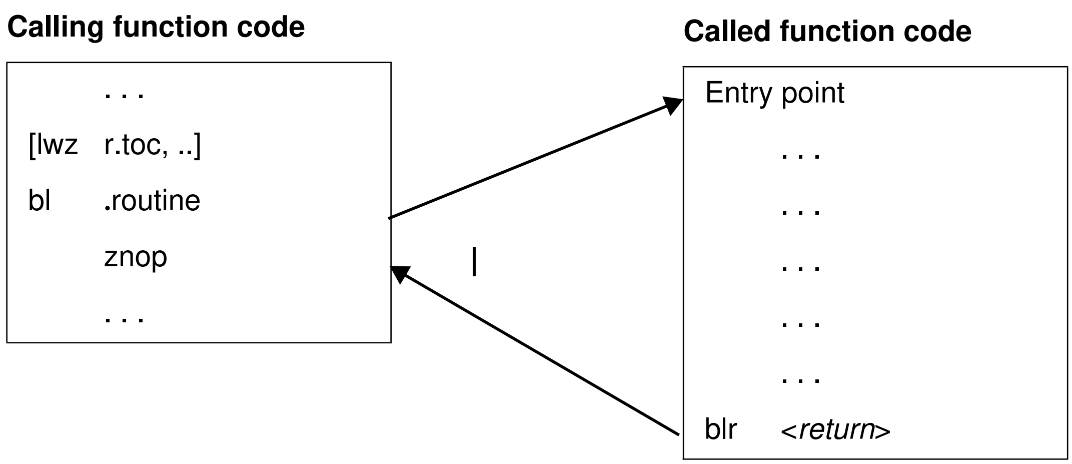

Figure 4 Calling sequence — call direct to local function

only needed if the TOC address is not already in register 2. Since r.toc is non-volatile, the called routine **must** insure that whatever value was in it at the time of the call is there at the time of the return. That might, of course, be achieved by never changing the value of r.toc during the procedure's execution.

If it is found at link time that the target is in fact not in the same executable, then the linker will insert "glue" code to implement the call, as described in section 3.6.4. The znop instruction following the branch and link in the caller is a placeholder for an instruction, inserted by the linker, to restore the TOC register from where the glue sequence has saved it. This arrangement allows a fairly efficient implementation of glue code since it avoids saving and restoring the link register. The placeholder need only be present if the possibility exists that the target of the call is "imported". It would never be needed, for example, if the target of the call were in the same source file as the caller.

If the executable is so large that the "bl" (branch and link relative) instruction generated for a local call cannot reach the target, the link will fail[1]. For such large programs, a successful link may only be achievable by arranging for some of the calls to be implemented via a function descriptor (see Section 3.6.2).

Function descriptors need not be generated for functions declared (C) "static" unless the address of such a function is taken, because static functions cannot be referenced from outside the compilation unit and intra-compilation calls to static functions do not involve a function descriptor.

3.6.2 Optional in-line external linkage via function descriptor

Linkage code that makes use of the function descriptor for the routine can be inserted in place of the default branch and link. If generated during compilation, such "external linkage" has the advantage that instructions for the call/return sequence are subject to optimizations like scheduling and elimination of redundant loads. When the function is called using its name, the address of the descriptor is fetched from the TOC, as shown in Figure 5. The new TOC register and target address are loaded from the descriptor, and a branch and link to the link register gets directly to the called routine. The exception processing protocol requires that a procedure that implements a call via a function descriptor save its incoming value of r.toc and link register by code in its prologue so that they can be properly restored. Unless required earlier in the procedure's code, the saved value of r.toc need not be restored to r.toc until the procedure is about to return.

3.6.3 Calls via function pointers

Since it is not in general known at compile time whether calls via a function pointer are to local or imported routines, the default implementation of such calls follows the same protocol as the optional in-line external linkage. The address of the function descriptor does not come from the TOC in this case, however; it is the value of the pointer variable. As is the case for procedures with compiled in-line external linkage calls, procedures with calls via pointers must save the incoming values of r.toc and the link register since they will both be modified in the course of executing the call. This must be done in such a way that stack unwinding mechanisms associated with exception processors can determine how to restore them. The details are system-specific.

3.6.4 Linker-introduced "glue" code

Compiler generated local calls to procedures that end up not in the same executable as their caller are handled in a special way. Such procedures occur, for example, in dynamically linked libraries.

In order to link a program that references procedures from a different executable, a special code sequence called *linkage glue* is introduced into the executable at link time. In addition, an entry is made in the executable's TOC that will point to the unique function descriptor in the "foreign" executable. The linkage glue has the same entry point name as the called function so the branch and link generated for the local call actually branches to the glue. For a call to foo, the general form of the glue is

```
.foo:
```

1. A linker function could be designed to enable "glue" code to be inserted to handle this problem.

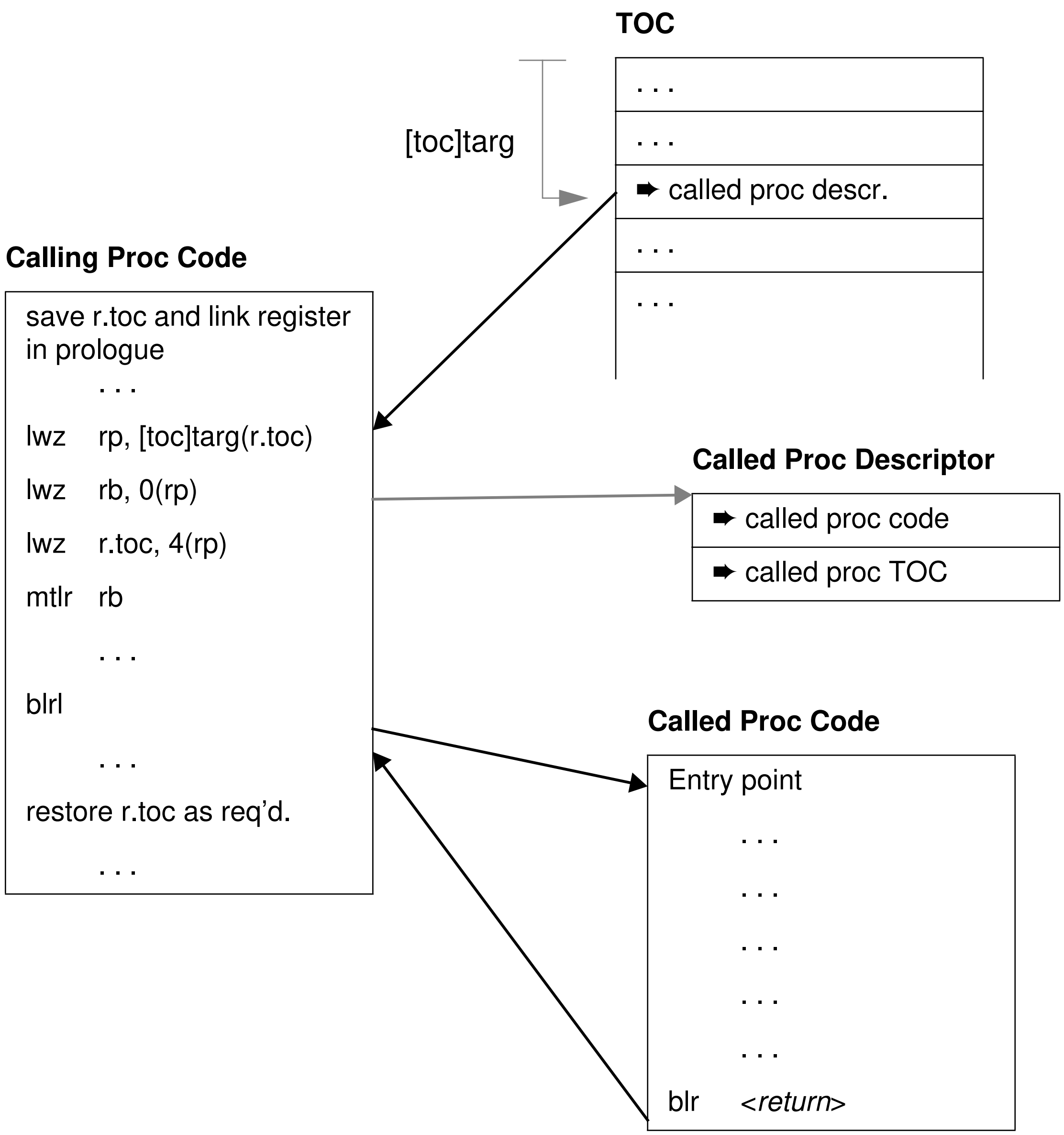

Figure 5 Calling sequence — call via function pointer from TOC

```
lwz     r.11, [toc]foo(r.2)        # get address of foo's descriptor
b       .ptrgl                     # branch to pointer glue
```

[toc]foo refers to the TOC entry that points to foo's descriptor. The special sequence, .ptrgl, is the same for all calls to "imported" functions:

```
.ptrgl:
        lwz     r.12, 0(r.11)          # get address of .foo
        stw     r.2, glsave(r.1)       # save the caller's TOC register (glsave=8)
```

```
mtctr   r.12            # get ready to branch to .foo
lwz     r.2,4(r.11)     # get callee's TOC
bctr                    # branch  to .foo
```

The offset glsave (=8) is the fixed offset from the caller's stack pointer where the glue can save r.2. The *branch-and-link* target is placed in the count register to avoid changing the link register in the glue code itself. Note that the glue code cannot itself acquire a stack frame since the called procedure expects to find its arguments at known offsets from its caller's stack pointer and, for this purpose, the glue code is not its caller!

In addition to introducing the specialized glue sequence, the linker must also replace the znop instruction with

```
lwz     r.2,glsave(r.1)
```

so that the caller's TOC register is properly restored. In AIX, the linker actually examines the word following the bl instruction to see if it is a znop. A better way is to provide an RLD to instruct the linker to perform this function. The RLD points to the location of the znop and identifies the function by its symbol table entry. This technique does not require a specific instruction to be architected to represent the znop, and does not require the compiler to place it immediately following the bl.

Appropriate information must be left around after linking to enable the loader to fill in the TOC entry for the descriptor.

Although scheduled as well as possible, the glue code will be slower than if the compiler had generated the equivalent in-line linkage. This should always be considered for critical applications. Therefore, even though the .ptrgl sequence could be used to implement calls through pointers, better performance is achieved by using the in-line sequence shown in Figure 5.

3.6.5 Lazy loading

Lazy loading is the term given to the support for loading a function at run time, the first time that it is called. This is a performance enhancement for programs that contain references to functions that are called only rarely, or perhaps never in some environments. No overhead for loading them occurs at initial program load, and even if they are never available, an error will occur only if they are actually called.

The details of support for this function are system specific.

3.6.6 Pragmas for controlling procedure linkage

By default, most compilers assume that local linkage is to be generated for all calls not to function pointers. As has been described, calls to imported functions are implemented by linker-introduced glue sequences. Somewhat greater efficiency can be achieved if these sequences are "inlined". The Appendix, Section 8.1 shows how options and #pragma stattements can be used in the AIX compilers to accomplish such inlining.

3.6.7 Parameter passing[1]

For a RISC machine such as PowerPC, it is generally more efficient to pass arguments to the parameters of called functions in registers (both general registers and floating-point registers)

1. Please refer to section 5.4 for a discussion of parameter passing in the NT implementation as it is somewhat different from what is presented here.

than to construct an argument list in storage or to push them onto a stack. All computations must be performed in registers anyway, and memory traffic can be eliminated if the caller can compute arguments into registers, pass them in the same registers to the called function, and the called function can use them for further computation in those same registers.

3.6.7.1 Location of arguments

For Little-Endian PowerPC, as for AIX on Power, we pass up to eight words in the general registers, loading them sequentially into general registers 3 through 10. In addition, up to thirteen floating point arguments can be passed in floating point registers 1 through 13. If fewer (or no) arguments are passed, the unneeded registers are not loaded and will contain undefined values on entry to the called function.

In only two cases, both relatively infrequent, must arguments be in storage for a call:

- When the amount of data being passed is more than will fit in the eight general registers (and the thirteen floating point registers) provided. The remainder is placed in storage.
- When the called function takes the address of one or more of its input parameters. For example, printf() does this when it interprets at run time the types of its parameters, and steps through them one at a time using the varargs macro.

For the first case, the excess values are passed in an area of the caller's stack frame, beginning at a fixed location from the stack frame (16 bytes). The compiler reserves in the caller's stack frame enough space for the caller to construct the longest overflow argument list necessary for all the functions that it calls. Figure 6 shows a detail of the **caller's** stack frame where space is reserved

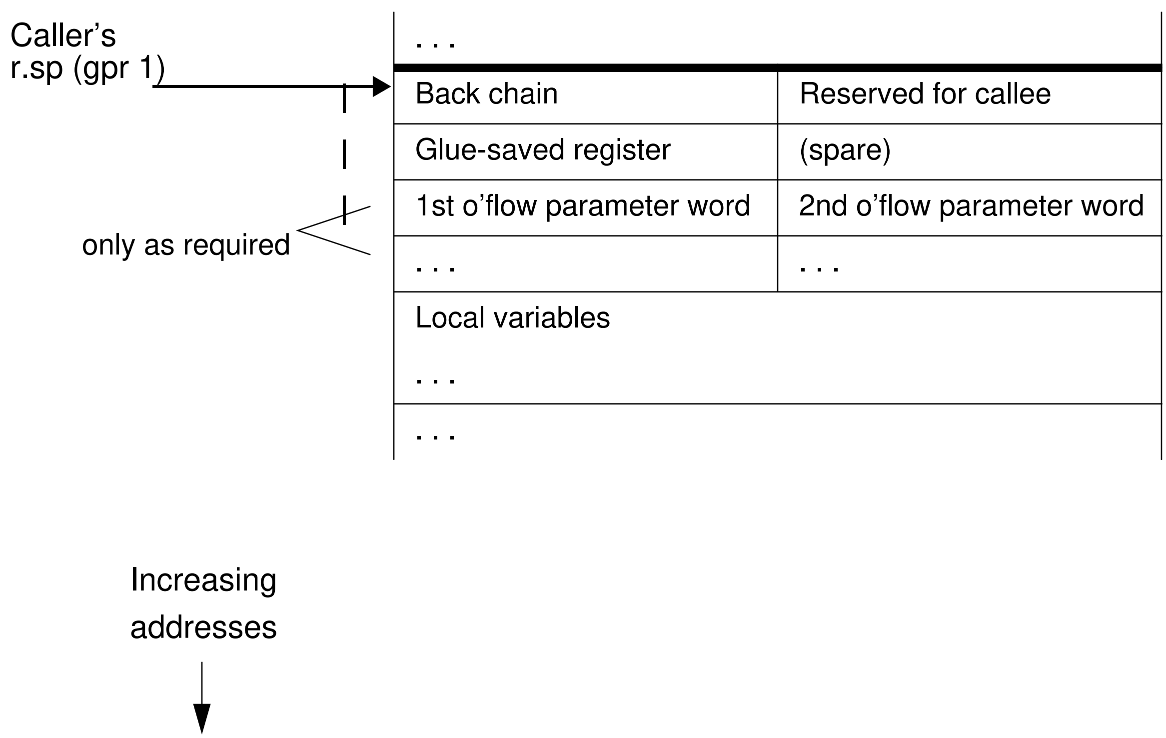

Figure 6 Parameter list area in stack frame

for the excess (over 8 words) of arguments passed to a called function's parameters. The called function accesses the first eight words of its parameters in general registers 3 through 10. For parameters beyond the eighth word, it must access the list constructed in the caller's stack frame. It is trivial to locate this: the compiler knows the length of the stack frame that it has built, and the list starts at a fixed location in its caller's stack frame as shown above. Thus the ninth parameter word is at offset 16, the tenth at offset 20, etc. Alternatively, since the caller's stack pointer is always stored at offset 0 from the current stack pointer, that value can be loaded and used to address the parameter list.

In the case where the called function takes the address of an input parameter, or treats its parameters as an array, it may be necessary to materialize the entire argument list in a contiguous area of storage. To achieve this, the called function allocates additional space on the stack to hold the first eight parameter words. This space is allocated by moving the calling function's stack frame header down 32 bytes, thus providing 32 bytes adjacent to the argument list overflow area. The compiler can use the eight words thus allocated to store the parameter registers as required to complete the argument list in storage[1], as shown in figure Figure 7.

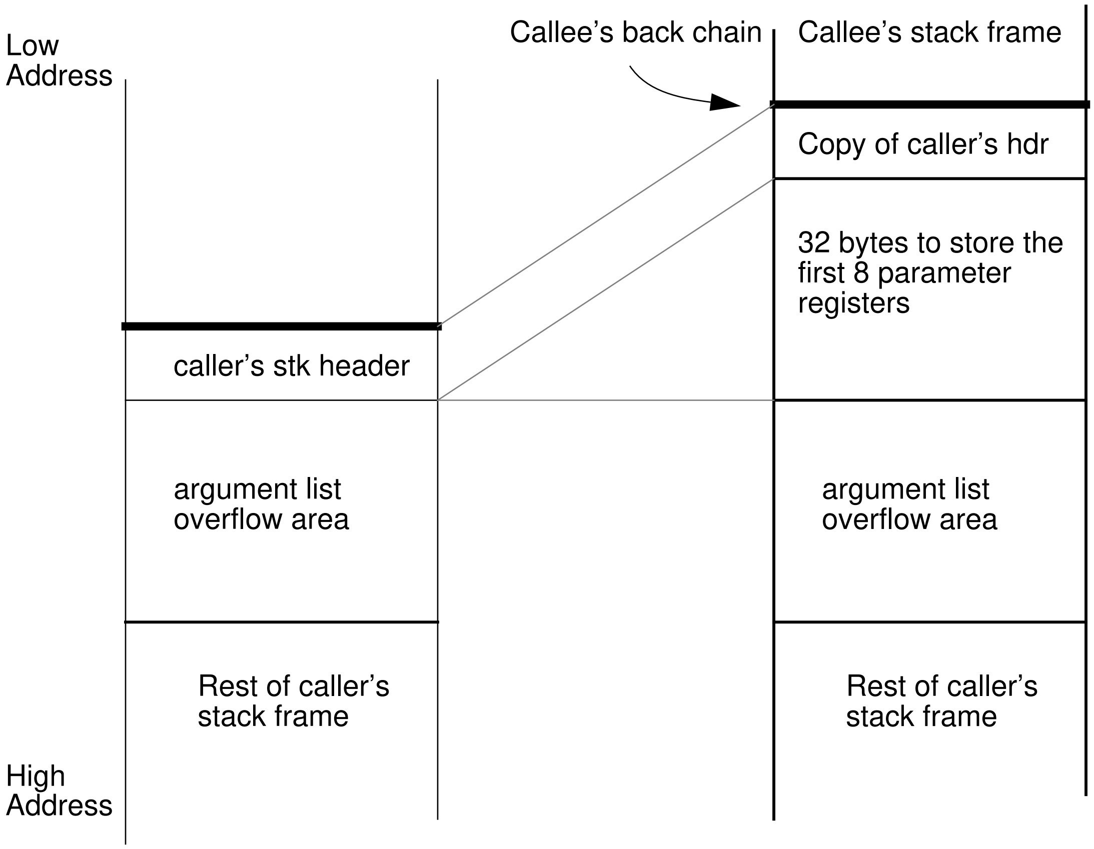

Figure 7 Stack frame showing how callee copies the caller's header making room for the first 8 parameter registers contiguous with other arguments

1. Since the callee's back chain now points to a different address than did the caller's stack pointer prior to the call, programs such as the debugger must ensure that they address the caller's automatic variables from a base other than the caller's stack frame pointer. An appropriate such base is the caller's *establisher frame*: i.e., the caller's caller's stack frame pointer.

The allocation and copy is part of the callee's prologue, and must, of course, be undone as part of its epilogue. The exact sequences to accomplish this are designed so that exceptions that occur at inconvenient times will be processed correctly. Typical sequences are shown in Section 3.6.8.5.

The advantage of the scheme described here is that the overhead for its implementation occurs only when a function addressing its parameter list is actually executed. For all other functions, there is no penalty in either space or time.

3.6.7.2 Floating point arguments

Floating point arguments are passed in floating point registers 1 through 13. If there are more than thirteen such arguments, the remainder are passed in storage in the argument extension area, intermixed with any non-floating arguments as indicated in Section 3.6.7.3.

3.6.7.3 Mapping the argument list

The details of argument passing are most easily understood by considering first how the arguments would appear in storage if they were all passed that way. Then, the actual situation is derived from this conceptual argument list.

Assume that the conceptual argument list begins on a doubleword boundary. Now take all the arguments in order, including aggregate arguments passed by value (e.g., structures in C) and put them sequentially in the list, observing alignment requirements. These requirements are essentially the natural alignment of the argument, but not smaller than fullword. The result is the conceptual argument list, in which each argument appears exactly as it would appear in storage and each separate argument begins on *at least* a word boundary.

What is actually passed is like the conceptual argument list except that some of the arguments appear in registers rather than in storage. Exactly what happens depends on the existence and kind of a function prototype for the called function.

In the absence of a function prototype, the contents of the first eight fullwords (including any floating point or aggregate values) are placed in general purpose registers 3 through 10 rather than in storage. Additionally, the first thirteen floating point *scalar* arguments are placed in floating point registers 1 through 13 (despite the fact that they may already appear in a register or in storage). That part of the conceptual argument list containing values after the eighth fullword appears in the caller's stack frame beginning at offset 16 from the caller's stack pointer..

If a function prototype is present specifying all the parameters of the function[1], redundancy can be eliminated from the argument list: floating point values that appear in floating point registers need not also appear in general purpose registers or in storage. However, even in this case the space for them is retained. This is necessary because the called function must be able to access its parameters the same way whether or not a function prototype was available to its caller..

If a function prototype is present and contains an ellipsis, arguments appearing before the ellipsis are treated as if a function prototype were present; all others are treated as if there were no function prototype.

---

1. It contains no ellipsis, so no call instruction to the function supplies more arguments than there are parameters in the prototype. In effect, the function and its caller agree in advance about the number and types of parameters. This would not be true, for example, for a function like printf, which determines its parameter types at run time and uses the varargs macro or direct address arithmetic to access them.

3.6.7.4 Materializing the parameter list in storage

The parameter passing protocol described in section 3.6.7.3 permits a rather general implementation of functions that access parameters in a variety of ways by means of pointers. The key point is that there is sufficient information to permit the compiler to materialize the entire conceptual argument list in storage.

If a function takes the address of any of its parameters, it is the compiler's responsibility to ensure that the proper value exists in every part of the argument list that may be accessed via a pointer derived from that parameter address. This is guaranteed if the function's prologue contains code to store to their home locations all parameters that are passed in registers. Even if the number and types of the arguments that are passed to a function are not known at compile time, it is always safe to generate code to store the eight general purpose registers, and, following this, any floating point registers that are explicitly mentioned in parameter declarations.

Of course some optimization is possible. It is not really necessary to store a general purpose parameter register if a floating point registers will be stored to the same location. More signifi-canly, there are cases in which analysis by the compiler would indicate that not all of the argument list need be materialized; but in the absence of such a result, the address of any parameter must be treated as a potential alias of every part of the argument list, even parts of it that are not explicitly mentioned as parameters..

3.6.7.5 Return values

Scalar non-floating data are returned in general purpose registers, and scalar floating values are returned in floating point registers.

Data that are not returned in registers are returned in storage, the address of which is passed as a (hidden) first argument by the caller.

3.6.7.6 Some examples of parameter passing

Assume the following declarations:

```
typedef struct { int a,b,c; double dd;} sparm;
        /* assumed structure mapping requires dd to be doubleword aligned */
sparm s,t;
int x, *y;
char c;
double ff,gg;
```

Example 1:

```
x = noProto(x,c,y,ff,s,gg)
```

The indicated arguments would appear in the conceptual argument list, which begins at offset -16 (-10 hex) from the caller's stack pointer, as follows: x at offset -16 (-10 hex), c at offset -12 (-C hex), y at offset -8 (-8 hex), etc. Because of alignment requirements, ff begins at offset 0 (0 hex), skipping the word at offset 36 (24 hex). Hence the actual places occupied by the arguments are as indicated in Figure 8.

gprs		offset from caller's stk ptr		fprs	
gpr3	x	-10 (hex)			
gpr4	c	-C			
gpr5	y	-8			
gpr6		-4			
gpr7	ff(low)	0		fpr1	ff
gpr8	ff(hi)	4			
gpr9	s.a	8			
gpr10	s.b	C			
		10 (hex)	s.c		
		14			
		18	s.dd(low)		
		1C	s.dd(hi)		
		20	gg(low)	fpr2	gg
		24	gg(hi)		

Figure 8 Parameter passing, example 1

Example 2.

```
sparm Proto(sparm,double, int,double);
t = Proto(s,ff,x,gg);
```

Since a structure value is being returned, the caller allocates temporary space for it in its stack frame and passes the address of the space as a (hidden) first argument in register r.3.[1] The required alignment for the argument structure s forces it to begin in register r.5 (associated with offset -8 (-8 hex) rather than r.4. The presence of the prototype makes it possible to pass the floating arguments in floating point registers only. Space for them must still be reserved, however, which explains why x appears at offset 24 (18 hex) rather than 16. The arguments thus actually appear as shown in Figure 9.

gprs		offset from caller's stk ptr		fp's	
gpr3	&temp	-10 (hex)			
gpr4		-C			

Figure 9 Parameter passing, example 2

1. When possible, an optimizing compiler might be able to use the target of the assignment (t) directly as the temporary, thus avoiding a copy when the function returns.

gprs		offset from caller's stk ptr		fp's	
gpr5	s.a	-8			
gpr6	s.b	-4			
gpr7	s.c	0			
gpr8		4			
gpr9	s.dd(low)	8			
gpr10	s.dd(hi)	C			
		10		fpr1	ff
		14			
		18	x		
		1C			
		20		fpr2	gg
		4c			

Figure 9 Parameter passing, example 2

Example 3:

This example is rather simple and is included to show the basic efficiency of the parameter passing.

```
extern double foo(int *,double,int,int,double,double)
ff = foo(y,ff,x,*y,gg,gg+1.0);
```

y is in r.3, ff is in f.1, x is in r.7, *y is in r.8, gg is in f.2, and gg+1.0 is in f.3. The double is returned in f.1. Figure 10 diagrams the layout. If the function prototype were not present, in addition to

gprs		offset from caller's stk ptr		fprs	
gpr3	y				
gpr4					
gpr5				fpr1	ff, return val
gpr6					
gpr7	x				
gpr8	*y				
gpr9				fpr2	gg
gpr10					
		10 hex		fpr3	gg+1.0
		14 hex			

Figure 10 Parameter passing, example 3

the above, the argument ff would also be in r.5 and r.6, gg would also be in r.9 and r.10, and gg+1.0 would also be in the double word of storage at offset 16 (10 hex) from the stack pointer. Notice that the presence of the prototype *allows* for a more economical passing strategy; it does not *force* it: no harm is done if the floating arguments appear in both places.

3.6.8 Register saving/restoring

To avoid the code expansion that would result from placing long strings of loads and stores in-line, entry points are defined for routines that save or restore particular sets of general or floating-point registers as well as the other registers (LR, etc.) involved in the calling sequence.

If fewer than three general registers or floating-point registers need to be saved or restored, the compiler should generate in-line code to do so since that will take less time than calling a save or restore subroutine, although the routines provided will save/restore as few as one register. The breakpoint between in-line save/restore and calling a save/restore routine need not be fixed at three registers; if the compiler can generate in-line save/restore sequences that are faster and not substantially more bulky than the calls to the save/restore routines, it may do so.

3.6.8.1 Millicode for saving/restoring registers

<u>GPR save routine:</u>

On entry, r.12[1] points to end of area in which to save GPRs.

1. If no floating registers need be saved, r.12 is of course set to r.sp. A special save (and restore) millicode routine could be defined to use r.sp directly. That would save two cycles in the linkage as r.12 would not then be used. Whether or not such millicode will be supplied for all environments is TBD.

```
._savegpr_13:     stw     r.13, -76(r.12)
._savegpr_14:     stw     r.14, -72(r.12)
._savegpr_15:     stw     r.15, -68(r.12)
._savegpr_16:     stw     r.16, -64(r.12)
._savegpr_17:     stw     r.17, -60(r.12)
._savegpr_18:     stw     r.18, -56(r.12)
._savegpr_19:     stw     r.19, -52(r.12)
._savegpr_20:     stw     r.20, -48(r.12)
._savegpr_21:     stw     r.21, -44(r.12)
._savegpr_22:     stw     r.22, -40(r.12)
._savegpr_23:     stw     r.23, -36(r.12)
._savegpr_24:     stw     r.24, -32(r.12)
._savegpr_25:     stw     r.25, -28(r.12)
._savegpr_26:     stw     r.26, -24(r.12)
._savegpr_27:     stw     r.27, -20(r.12)
._savegpr_28:     stw     r.28, -16(r.12)
._savegpr_29:     stw     r.29, -12(r.12)
._savegpr_30:     stw     r.30, -8(r.12)
._savegpr_31:     stw     r.31, -4(r.12)
                  blr                                # return
```

GPR restore routine:

On entry, r.12 points to end of area in which GPRs were saved.

```
._restgpr_13:     lwz     r.13, -76(r.12)
._restgpr_14:     lwz     r.14, -72(r.12)
._restgpr_15:     lwz     r.15, -68(r.12)
._restgpr_16:     lwz     r.16, -64(r.12)
._restgpr_17:     lwz     r.17, -60(r.12)
._restgpr_18:     lwz     r.18, -56(r.12)
._restgpr_19:     lwz     r.19, -52(r.12)
._restgpr_20:     lwz     r.20, -48(r.12)
._restgpr_21:     lwz     r.21, -44(r.12)
._restgpr_22:     lwz     r.22, -40(r.12)
._restgpr_23:     lwz     r.23, -36(r.12)
._restgpr_24:     lwz     r.24, -32(r.12)
._restgpr_25:     lwz     r.25, -28(r.12)
._restgpr_26:     lwz     r.26, -24(r.12)
._restgpr_27:     lwz     r.27, -20(r.12)
._restgpr_28:     lwz     r.28, -16(r.12)
._restgpr_29:     lwz     r.29, -12(r.12)
._restgpr_30:     lwz     r.30, -8(r.12)
._restgpr_31:     lwz     r.31, -4(r.12)
                  blr                                # return
```

FPR save routine:

On entry, r.sp points to the end of the area in which FPRs are to be saved.

```
._savefpr_14:    stfd    f.14, -144(r.sp)
._savefpr_15:    stfd    f.15, -136(r.sp)
._savefpr_16:    stfd    f.16, -128(r.sp)
._savefpr_17:    stfd    f.17, -120(r.sp)
._savefpr_18:    stfd    f.18, -112(r.sp)
._savefpr_19:    stfd    f.19, -104(r.sp)
._savefpr_20:    stfd    f.20, -96(r.sp)
._savefpr_21:    stfd    f.21, -88(r.sp)
._savefpr_22:    stfd    f.22, -80(r.sp)
._savefpr_23:    stfd    f.23, -72(r.sp)
._savefpr_24:    stfd    f.24, -64(r.sp)
._savefpr_25:    stfd    f.25, -56(r.sp)
._savefpr_26:    stfd    f.26, -48(r.sp)
._savefpr_27:    stfd    f.27, -40(r.sp)
._savefpr_28:    stfd    f.28, -32(r.sp)
._savefpr_29:    stfd    f.29, -24(r.sp)
._savefpr_30:    stfd    f.30, -16(r.sp)
._savefpr_31:    stfd    f.31, -8(r.sp)
                 blr                          # return
```

<u>FPR restore routine:</u>

On entry, r.sp points to the end of the area in which FPRs were saved.

```
._restfpr_14:    lfd     f.14, -144(r.sp)
._restfpr_15:    lfd     f.15, -136(r.sp)
._restfpr_16:    lfd     f.16, -128(r.sp)
._restfpr_17:    lfd     f.17, -120(r.sp)
._restfpr_18:    lfd     f.18, -112(r.sp)
._restfpr_19:    lfd     f.19, -104(r.sp)
._restfpr_20:    lfd     f.20, -96(r.sp)
._restfpr_21:    lfd     f.21, -88(r.sp)
._restfpr_22:    lfd     f.22, -80(r.sp)
._restfpr_23:    lfd     f.23, -72(r.sp)
._restfpr_24:    lfd     f.24, -64(r.sp)
._restfpr_25:    lfd     f.25, -56(r.sp)
._restfpr_26:    lfd     f.26, -48(r.sp)
._restfpr_27:    lfd     f.27, -40(r.sp)
._restfpr_28:    lfd     f.28, -32(r.sp)
._restfpr_29:    lfd     f.29, -24(r.sp)
._restfpr_30:    lfd     f.30, -16(r.sp)
._restfpr_31:    lfd     f.31, -8(r.sp)
                 blr                          # return
```

Additional sequences can, of course, be provided. For example, if a sequences to save and restore gprs based on r.sp were provided, a compiler could use those in place of the ones based on r.12 to save and restore general purpose registers for the common case where no floating point non-volatile registers are required to be saved. That would eliminate having to set a value into r.12 prior to branching to the sequence.

3.6.8.2 Prologue/epilogue: saving only general registers

Assume that our called routine needs to save *n* general purpose registers. A typical prologue and epilogue for our routine might look like this:

```
.routine:
        mflr    r.0                         # get LR into reg 0
        mr      r.12, r.sp                  # point to GPR save location
        bla     ._savegpr_32-n[1]           # save GPRs
        stw     r.0, LRSAVE(r.sp)           # save LR contents
          (save CR, r.toc if necessary)
        stwu    r.sp, –framelength(r.sp)    # set back chain, new stack pointer
        . . .
        . . .                               # body of function
        . . .
        la      r.12, framelength(r.sp)     # point to caller's stack frame
                                            # (reload from chain or compute from
                                            # frame pointer if alloca() used)
        lwz     r.0, LRSAVE(r.12)           # reload saved LR contents
          (reload saved CR, r.toc if necessary)
                                            # r.12 already points to GPR save location
        mtlr    r.0                         # move LR contents back into LR
        mr      r.1, r.12                   # restore stack register[2]
        ba      ._restgpr_32-n              # restore GPRs and return
```

The GPR save routine saves only the GPRs. All else is done in-line: we move the LR into r.0 because it will be destroyed by the branch-and-link-absolute, we compute the location where the GPRs will be stored (not necessarily adjacent to the stack frame header), we store the LR contents from r.0, and we compute the new stack pointer and store the stack chain (in one atomic operation). The link register value is saved at offset LRSAVE, which could be chosen to be -4*(n+1), or alternatively +4, using the reserved word in the caller's stack frame header. On exit, these actions are reversed. Since r.2 was never changed, the value in it on exit is the same as on entry, as required.

3.6.8.3 Prologue/epilogue: saving both general and float registers

Assume that our called routine needs to save *m* floating point registers and *n* general purpose registers. The prologue and epilogue of our routine might look like this:

```
.routine:
        mflr    r.0                         # get LR into reg 0
        subi    r.12, r.sp, 8*m             # point to GPR save location
        bla     ._savegpr_32-n              # save GPRs
        stw     r.0, LRSAVE(r.sp)           # save LR contents
        bla     ._savefpr_32-m              # save FPRs
          (save CR if necessary)
        stwu    r.1, –framelength(r.sp)     # set back chain, new stack pointer
```

1. "32-n" is not intended to denote an expression recognized by any assembler. It is merely part of an external label name. If n is 5, for example, the code would read "._savegpr_27".,
2. The stack pointer is restored just prior to the branch to the gpr restore millicode in order to satisfy exception handling requirements related to asynchronous events (see section 3.10.2 and 5.7.6).

```
        . . .
        . . .                                   # body of function
        . . .
        la      rx, framelength(r.sp)           # point to caller's stack frame
                                                # (reload from chain if alloca() used)
          (reload saved CR if necessary)
        lwz     r.0, LRSAVE(rx)                 # reload LR value from stack frame
        subi    r.12, rx, 8*n                   # point to GPR save location
        bla     ._restgpr_32-n                  # restore GPRs
        mtlr    r.0                             # put reloaded value back into LR
        mr      r.1, rx                         # restore stack pointer
        ba      ._restfpr_32-m                  # restore FPRs and return
```

In this case, LRSAVE could be chosen to be -4*(n+1)-8*m, or (again) +4 as in the previous example. If r.toc or the condition register need to be saved, the save locations could be chosen adjacent to the last saved non-volatile register. Observe that r.toc *cannot* be saved in the stack frame header of either the caller or the callee: that location is only for use by glue code. Of course, more economical use of registers would be achieved by making r_x = r.12, and replacing the mr instruction by addi r.1,r.12,8*n.

3.6.8.4 Prologue/epilogue: saving only float registers

Assume that our called routine needs to save m floating point registers. The prologue and epilogue of our routine might look like this:

```
.routine:
        mflr    r.0                             # get LR into reg 0
        bla     ._savefpr_32-m                  # save FPRs
        stw     r.0, LRSAVE(r.sp)               # save LR in stack frame
          (save CR, r.toc if necessary)
        stwu    r.sp, –framelength(r.sp)        # set back chain, new stack pointer
        . . .
        . . .                                   # body of function
        . . .
          (reload saved CR, r.toc if necessary)
        lwz     r.0, LRSAVE+framelenth(r.sp)# pick up LR from stack frame
        la      r.sp, framelength(r.sp)         # point to caller's stack frame
                                                # (reload from chain if alloca() used)
        ba      ._restfpr_32-m                  # restore FPRs
```

Note that the old value of the link register is loaded from its saved location prior to the restoration of the stack frame, so that the offset in the instruction has been adjusted accordingly.

3.6.8.5 Prologue/epilogue: a function that addresses its parameters

The following is a typical prologue:

```
.printf:
        mflr    r.0                             #prepare to save link register
        lwz     r.11,0(r.1)                     #copy caller's stack header
        lwz     r.12,8(r.1)
        stw     r.11,-32(r.1)
```

```
        lwz     r..11,12(r.1)
        stw     r.12,-24(r.1)
        stw     r.11,-20(r.1)
        stw     r.0,-28(r.1)                #save link register in caller's header
        addi    r.1,r.1,-32
        stwu    r.1,r.1,-framesize.printf   #buy stack frame for printf
        bl      register_save_millicode     #save used non-volatile registers
        stw     r.3,16(r.1)                 #"blindly" store parameter gprs
        stw     r.4,20(r.1)
        ...
        stw     r.10,44(r.1)
printf.body:
```

Note that at every point, r.1 points to a header with the correct values in it, as required for correct processing of exceptions. Also, at the end of the prologue, the complete parameter list appears in contiguous storage, no matter how many arguments were actually passed.

The epilogue code corresponding to this is:

```
printf.epil:
        lwz     r.0,framesize+4(r.1)        #prepare to restore link register
        lwz     r.11,framesize(r.1)         #copy header back
        mtlr    r.0                         #restore link register
        lwz     r.12,framesize+8(r.1)       #continue to copy header back
        stw     r.11,framesize+32(r.1)
        lwz     r.11,framesize+12(r.1)
        stw     r.12,framesize+40(r.1)
        stw     r.11,framesize+44(r.1)
        addi    r.1,r.1,framesize+32        #restore caller's stack pointer
        blr                                 #return
```

These sample prologues and epilogues indicate possible sequences that a compiler might generate. They are actually not part of the linkage conventions proper, and in fact a compiler might be able to obtain better performance by performing the instructions in a different order or including in the sequence other instructions from the program proper. In order for exception processing in most systems to operate properly, it is necessary that values in non-volatile registers can be restored by the exception processing code. Thus, sufficient information must be available at run time to accomplish this task irrespective of where in the instruction stream the exception occurred. The protocols used to achieve this are system dependent. For further details, see the discussion under exception processing.

3.7 Effect of alloca() on stack frame

In C, the alloca() function allocates storage in the stack frame for local non-static variables whose sizes are not known at compile time. The function is typically built into the compiler and is executed as in-line code rather than as a true call. On Little-Endian PowerPC, we perform the alloca() function by bumping the stack pointer back (toward lower addresses) to make room for the allocated variable and readjust some of the stack frame header information.

Figure 11 shows what happens to the stack when alloca() is issued to obtain storage to hold variable x. Note that r.sp still points to the current stack frame, as it always must, and that an addi-

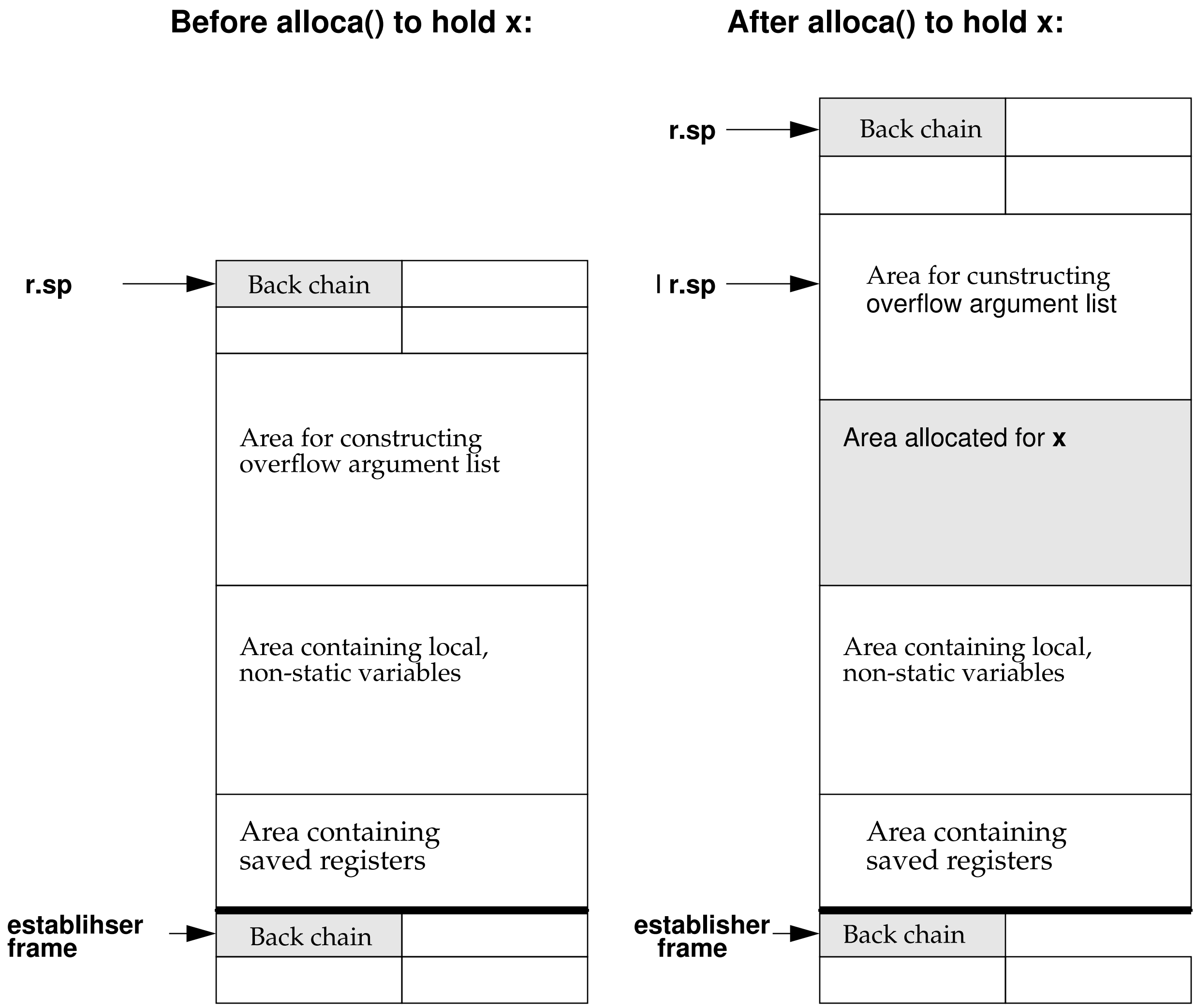

Figure 11 Stack frame before and after alloca()

tional register is used to address that portion of the stack frame containing ordinary (non-alloca()) local variables. Here we have called the additional register r.frame, for "frame pointer", but this value is not pre-assigned to a specific register and can vary within a function as the compiler's register allocation mechanism finds best.

When variable x is allocated, the stack pointer (r.sp) is decreased by the length of x (rounded up to satisfy stack frame alignment requirements). The address of the previous stack frame, (the back chain) is stored in the first word at the new header location. None of the other contents of the stack frame header are copied to the new header location, because the other header cells are never used by the current function. If the compiler has assigned a register to provide the base address for references to the function's local non-static, non-alloca() variables, the value in that register does not change. The compiler is free to choose whatever value it wants for this purpose, but a convenient value would be the stack pointer of the caller: the *establisher frame*.

The implementation of alloca() must insure that the stack does not overflow. The same considerations apply here as for the case when a stack frame is allocated. The guard page will automatically detect stack overflow when r.sp is updated. If the amount to be allocated exceeds the size of the guard page, an explicit test must be made like the one made when a similarly large stack frame is allocated.

3.8 Other built-in subroutines

In addition to the register saving and restoring functions and routines to allocate large objects on the stack, certain other heavily used functions have "well known" entry names that are never exported and so must be bound directly with their caller. This avoids the necessity for accessing these functions via the TOC.

This rapid linkage is intended for high-use routines such as those for character string operations, math operations that may have different underlying implementations on different processors, and the like.

Such routines are **not** passed their TOC pointer in register 2; they receive whatever the caller left there instead (since the linkage is done as if it were local). Linkage to such routines may be non-standard in other ways as well — they may be known to the compiler to kill fewer registers than usual, for example.

The list of such routines and their interface definitions is system specific.

3.9 Additional data structures for programs

In addition to the ".text" and ".data" sections that contain the executable code and the data that the code uses, additional data structures must be generated by compilers to support functions such as exception handling, traceback, and debugging. Since some of these are system specific, the sequel discusses them in a rather general fashion.

3.9.1 Tag Tables

Static information about procedures is used by RTE[1] routines to perform functions such as generating traceback information or processing user exceptions. In order to perform these functions, the system routines must be able to find the required information. In the AIX architecture, the information is located in what are called *tag tables*. Each procedure has a tag table which is located immediately after the code, and separated from it by a word of all 0 bits. Given an instruction address, the tag table for the procedure containing the instruction can be found by scanning forward for the word of 0's, which is a distinctive pattern. That location turns out to be poor because it introduces non-instruction data into the instruction cache, reducing its efficiency.

For Little-Endian PowerPC, the tag tables will be disjoint from the code and can be found using a *tag table index*, also disjoint from the code. Every executable or dynamic link library contains a single tag table index: an array of entries each of which contains pointers to the beginning and end of the code for a procedure. System exception handlers access a procedure's tag table to obtain static information needed to perform such functions as stack unwinding, tracebacks, or specialized exception processing. Thus, a tag table is required for every procedure for which any such information is needed. Clearly, this requirement is system specific, although some language

1. Run Time Environment

constructs (e.g., "try-except" in C) impose requirements of their own. A system function (TBD) will, given an instruction address, return the address of the relevant tag table index.

The entries in the tag table index are arranged so that the pointers to procedure beginnings appear in ascending order. Therefore, given a pointer to an instruction (i.e., an instruction address), a binary search over the table can be used to find the entry that corresponds to the procedure containing the instruction. If no entry corresponds to the procedure, that fact is also determined.

In the preceding paragraphs, the term *pointer* rather than *address* is used to emphasize that the values need not be actual addresses but rather values from which addresses can be determined; i.e, offsets from a known address. Since the tag table index can be in read-only storage, the address of the tag table index itself could be used as the known address. Making pointers relative in this way avoids the necessity of relocating the tag table index at load time.

Language processors will be responsible for producing sections containing tag table index entries and sections containing tag tables. The linker concatenates the tag table index entries in the same order as it concatenates the corresponding code sections, and locates the resulting index and the aggregated tag tables in read-only portions of the executable.

3.9.2 Tag table index format

Although the specific format is system dependent, the minimal information that must be included in the tag table index is the entry point address of the code corresponding to each procedure that has a tag table. If a part of the tag table is constant format, then that portion also can be contained in the tag table index.

3.9.3 Tag table format

This again is system specific. If not included in the tag table index, then a pointer to the end of the procedure must be included in the tag table so that, given an instruction address, its containing procedure can be obtained unambiguously (recall that not all procedures necessarily have associated tag tables).

For language processors derived from the AIX compilers, the following tag table format is derived from what is now used on AIX and can serve as a starting point for defining appropriate tag tables:

```
struct TAG_TBL
        {
            char        format=1;        // identifies format of table
                                   // the following is a table of format 1.
            char        lang_ident;      // Source language identifier (C=0,Fort=1,
                                         // Pas=2,Ada=3,PL/I=4,Basic=5,Lisp=6,
                                         // Cobol=7,Modula2=8,C++=9,RPG=10,
                                         // pl.8=11,ASM=12)
            unsigned    is_out_prol:1;   // true if out of line prolog or epilog
            unsigned    short_tag;1;     // true if tag table is abbreviated
            unsigned    int_proc:1;      // true if this is an internal procedure
            unsigned    no_toc:1;        // true if procedure does not have a TOC
            unsigned    saves_cr:1;      // true if prologue saves condition register
            unsigned    saves_r2:1;      // true if r.2 saved in r.31
```

```
        unsigned  saves_lr:1 ;      // true if contents of link register changed
        unsigned  no_backch:1;      // true if proc does NOT store backchain
        unsigned  has_alloca:1;     // true if procedure contains alloca
        unsigned  has_imask:1;      // true if there is an interrupt mask in table
        unsigned  has_try_e:1;      // true if procedure contains try-except
        unsigned  has_try_f:1;      // true if procedure contains try-finally
        unsigned  split_prologue    // true if prologue not always executed
        unsigned  spares:3;
        unsigned  code_end;         // offset of procedure's end from its entry
        unsigned  frame_siz;        // extent of stack frame
        char      gp_save;          // number of first gpr saved (32 if none)
        char      fp_save;          // number of first fpr saved (32 if none)
        unsigned char delta_anks// offset to CTL_ANK from addr of tag table
        unsigned char delta_try     // offset to TEF_BLK from address of tag table
        char      nam_len;          // length of procedure name
        char      name[nam_len+1];     // procedure's name (null terminated)
    } ;

struct TEF_BLK
    {
        unsigned  num;              // number of entries for try blocks
    struct TEF_BLK_DATA{
        unsigned  begin_try;        // offset of try from proc entry point
        unsigned  length_try,       // length of try
        unsigned  begin_f;          // offset of filter or finally frm proc entry point
        unsigned  begin_t;          // offset of exception handler, or 0
              }   tefs[num];        // one for each entry
    };

struct CTL_ANKS
    {
        unsigned  num;              // number of ctl anchors
        unisgned  anks[num];        // offset in stack frame of each anchor
    };
```

Given a pointer to the tag table, p, and assuming that p->has_try_e | p->has_try_f is "true", the pointer to the TEF_BLK subtable is obtained by adding the value in p->delta_try to p[1]. The pointer to the CTL_ANKS subtable is obtained similarly using p->delta_anks. That subtable contains pointers to heap storage that is to be restored automatically on exit from a procedure, "destructors", and the like.

3.9.4 Special Routines (millicode)

In order to process exceptions properly, some of the millicode sequences must be recognized and treated in a special fashion. Entries in the tag table index for such sequences serve to identify them.

1. More formally, (struct TEF_BLK *)(p->delta_try + (int)p).

3.10 Programmed Exception Processing

It is profitable to consider exception processing as consisting of two parts: language-*independent*, and language-*dependent*. For systems such as AIX and NT, the model of execution is stack-frame based, and the language-independent portion of exception processing encompasses the notion of *stack unwinding*. The language dependent portion deals with particular constructs in a language, such as *try-except* and *try-finally* in (Microsoft extended) C.

3.10.1 Stack Unwinding

Simply stated, *unwinding a frame* is the process that, given an instruction address (pc value) and a state, produces the instruction address of the branch instruction that implemented the call to the function containing the given instruction address, and produces the state at the time of the branch. The term *state* in this context means the set of values in all the non-volatile registers. For languages whose run time environment includes a dynamic stack, the state also includes the relevant address (sp) of the stack frame active when an instruction is executed.

Two basic steps are involved in stack unwinding:

- Given a pc (value), identify the procedure containing the instruction
- Knowing the procedure, restore the state to what it was when the procedure was called.

The first step makes use of the tag table index. A binary search will determine whether there is an entry for the procedure. If there is no entry, then (barring any special handling) the state of the machine is presumed to be unchanged from when the instant procedure was called, and the pc of the branch instruction is presumed to be in the link register.

If there is an entry, then the entry identifies the corresponding tag table. There must be sufficient information in the table to permit the stack undwinder to know whether or not the link register has been saved, which non-volatile registers (including the TOC register) have been saved and where, and to indicate the size of the current stack frame. Obviously, this is sufficient to restore the state to what it was when the procedure was called.

3.10.2 Special considerations

Special circumstances that may occur during exception processing require special considerations. The details will vary from system to system depending upon the specifics of the various implementations, but the following must be kept in mind when designing mechanisms:

- At what point in the code an interrupt occurs
- Asynchronous interrupts
- Multiple exceptions
- General performance

The first deals with knowing precisely the state when an interrupt occurs. For example, it may be that different numbers of registers are saved (and restored) depending on what path of execution is taken through the program. So restoring registers can only be done if restrictions on how registers are saved (and restored) coupled with the information in the tag table is sufficient to reconstruct what it is necessary to do.

Asynchronous interrupts present problems because they may occur at points where it is difficult to determine the state of the code. For example, if such an interrupt occurs after the stack frame pointer has been restored but before the code has branched back to its caller, then the stack pointer does not correspond to the procedure determined to be executing from the tag index table. Code sequences should be constructed to enable proper determination of the state of computation when such an interrupt occurs.

Exception handlers may involve code running on the program stack, but even if they are running on separate stacks, care must be taken to deal with exceptions that occur during exception processing.

Finally, attention must be paid to the performance cost involved in the implementation of exceptions, particularly for the vast majority of code that does *not* experience an exception. It is better for exception processing to be somewhat costly when an exception occurs (by definition an exceptional event), rather than entailing a cost for all cases when no exception occurs. Although a useful programming tool, exceptions should not become ordinary coding practice.

3.10.3 Register save/restore millicode

If an exception occurs in register save millicode (recognizable by its tag table index entry), it is safe to use as the pc the value in the link register. Partially executed save millicode does not change any state.

If an exception occurs in register restore millicode (again recognizable by its tag table index entry), the proper unwind step is to complete execution of the millicode and then take as the pc the target of the final branch. This gives the correct answer whether or not it is in the procedure that branched to the millicode.

3.10.4 Glue code

Glue code need not have entries in the tag table index since the link register is not changed by such sequences[1]. However, since the load instruction that restores the TOC register actually appears in the calling function (refer to Section 3.6.4), information must be available to exception processing routines like the stack unwinder to enable them to recognize that the TOC register must be restored as a part of stack unwinding. One way is for the unwinder to examine the instruction pointed to by the link register and recognize the load instruction that restores r.2. Another way is to include the necessary information in the tag table. In the latter case, the compiler is free to locate the `znop` instruction as it sees fit to achieve better performance.

3.10.5 Considerations for language processors

Various exception mechanisms are defined in various languages. The architecture of the tag index table and the tag table is designed to accommodate their implementation. In addition, system support in the form of stack unwinders, state savers/restorers, etc., may be defined in some platforms.

The responsibility of a language processor is to produce the information that is required to implement the language's exception mechanism for use both by system code (the tag table index) and language specific code (the tag table).

3.11 Millicode

We use the term "millicode" for code that is higher level than microcode but not up to the level of general-purpose functions. Examples include the register save and restore sequences discussion in Section 3.6.8, glue linkage code sequences, stack overflow checking code, and the code used to

1. Note that the NT implementation requires such entries, viz. Section 5.7.7

access special purpose registers described in Section 3.11.4. Such code has the following characteristics:

- It is intended to supply the function of an instruction that may be missing in certain implementations, or it papers over differences in semantics for the same instruction on different implementations.
- It may use non-standard linkage. Fewer registers may be killed across the call than the linkage standard says, for example. A compiler could take advantage of this and keep values in registers that would be considered volatile for an ordinary call.
- It must not require a TOC pointer change. This implies:
  - It cannot call any other routines except other millicode.
  - It cannot have any static storage.
  - It cannot be exported, so that no glue is required to call it.

Millicode should not ordinarily be used for routines such as those that would appear in libc, but might be useful for lower-level routines *called* by libc routines. If a millicode routine were provided that performed the function of strcpy() for example, its implementation as millicode would preclude the user's replacing the library version with his own. Rather, millicode routines might be provided to perform the function of the Power string instructions, which are missing from Little-Endian PowerPC, and these routines might be called by the library strcpy() routine.

3.11.1 Location of millicode

Millicode is not mapped at any specific location; it is called by its published name.

3.11.2 Calling sequence for millicode

A call to a millicode routine follows the same calling sequence conventions as for ordinary calls, with a few exceptions.

- The stack frame pointer must be in r.sp (register 1) as always. If the called routine buys a stack frame, it must follow all the conventions regarding setting the back chain, etc.
- More or fewer registers may be killed across the call than for a standard call. This is subject to negotiation between the millicode implementor and compiler implementor.
- Parameters and return values may be passed in non-standard registers. Also subject to negotiation with the compiler implementor.
- The actual call (branch and link instruction) is generated as if the millicode routine were known to be local to the executable load module. That is, a branch and link relative instruction directly to the entry point is generated rather than a sequence of code that loads from a function descriptor and then branches via the Link Register.

3.11.3 Compiling calls directly to millicode

The TOBEY compiler for AIX can do string copies by

- Compiling a call to the strcpy() routine in libc, or
- Generating Power string instructions in-line.

The equivalent on Little-Endian PowerPC would be by

- Compiling a call to the strcpy() routine in libc, or

- Generating calls to millicode routines that perform low-level string operations similar to those of the Power string instructions.

A compilation option (command-line flag, #pragma, #define, . . .) should indicate which strategy to follow.

3.11.4 List of supplied millicode routines

A set of routines is provided for reading and writing the special-purpose registers of PowerPC, as shown in Figure 12. Since some of the operations performed may require supervisor state privilege, programs running in problem state must be written to avoid the routines using those operations.

The BAT (Block Address Translation) register read/write routines use the following structure parameter:

```
typedef struct _bat {
        unsigned long batu;        // upper BAT register
        unsigned long batl;        // lower BAT register
} bat;
```

Register	Function prototypes for reading and writing	
DAR	void *	**ppc_read_dar** ();
	void	**ppc_write_dar** (void *addr);
Decrementer	unsigned long	**ppc_read_dec** ();
	void	**ppc_write_dec** (unsigned long val);
DSISR	unsigned long	**ppc_read_dsisr** ();
	void	**ppc_write_dsisr** (unsigned long val);
EAR	unsigned long	**ppc_read_ear** ();
	void	**ppc_write_ear** (unsigned long val);
DBAT *n*	bat	**ppc_read_dbat** (int num);
	void	**ppc_write_dbat** (int num, bat val);
IBAT *n*	bat	**ppc_read_ibat** (int num);
	void	**ppc_write_ibat** (int num, bat val);
PVR	unsigned long	**ppc_read_pvr** ();
SDR 1	unsigned long	**ppc_read_sdr1** ();
	void	**ppc_write_sdr1** (unsigned long val);
SPRG *n*	unsigned long	**ppc_read_sprg** (int num);
	void	**ppc_write_sprg** (int num, unsigned long val);

Register	Function prototypes for reading and writing
SR *n*	unsigned long **ppc_read_sr** (int num);
	void **ppc_write_sr** (int num, unsigned long val);
SRR 0	void * **ppc_read_srr0** ();
	void **ppc_write_srr0** (void *addr);
SRR 1	unsigned long **ppc_read_srr1** ();
	void **ppc_write_srr1** (unsigned long val);
TB	unsigned long **ppc_read_tb** ();
	void **ppc_write_tb** (unsigned long val);
TBU	unsigned long **ppc_read_tbu** ();
	void **ppc_write_tbu** (unsigned long val);
XER	unsigned long **ppc_read_xer** ();
	void **ppc_write_xer** (unsigned long val);

Figure 12 Millicode prototypes for reading/writing Special Purpose Registers

3.11.5 Additional Millicode

Other millicode sequences are system specific. Included are specific routines to save and restore registers, check for stack frame overflow, and the like.

4 Local vs. Imported calls

4.1 Categories of calls

Function calls in C can be grouped into several categories, based on what is known about the target of the call at compile time:

Call via	Known location of target	Syntactic form
Pointer-to-function	—	(*fnptr) ();
Function name	Part of caller's executable	fn ();
	External to caller (e.g., in a DLL)	fn ();
	Unknown at compile time	fn ();

Figure 13 Call categories

The location of the called function relative to the calling function has a strong influence on the code generated for the call.

- Calls to functions that are part of the caller's executable can usually be done with a simple branch-and-link-relative (bl) instruction rather than a sequence of instructions that load the entry point address and branch-and-link via the Link Register (blrl).
- Calls to functions that are part of the caller's executable can use the caller's TOC pointer; they need not load a new TOC pointer from a function descriptor.

In the absence of explicit information about the location of a called function, the compiler must make an assumption as to whether the call is "local" (to a function that is part of the caller's executable) or "imported" (to a function that is outside the caller's executable). As is apparent from the table above, the syntactic form of the call alone can distinguish calls via function pointers (which are presumed to be imported) from calls via function name, but the calls via name cannot be further subdivided. The compiler must therefore "guess" (that is, apply its default choice) whether to generate local or imported linkage, and this choice cannot be proper for all calls. Judicious use of compiler command-line options and #pragma statements (see section 3.6.6) that specify local or imported linkage for particular calls or groups of calls can strongly affect the efficiency of the compiled code.

4.2 Call via pointer-to-function

A pointer-to-function is actually a pointer to a function descriptor, which contains the called function's entry point address and TOC address. The called function can be anywhere relative to the caller. In particular, the called function's TOC pointer may be different than the caller's TOC pointer or it may be the same. In principle, the compiler could generate code to determine at run time whether the TOC pointer must be loaded for such a call, but the code to do the test would take longer than just loading the TOC pointer unconditionally.

As a result, calls via pointer-to-function are compiled to save the caller's TOC pointer (if not already done so earlier in the program), load the caller's entry point address and TOC pointer, and branch-and-link via the Link Register.

4.3 Call via function name

Here the compiler has a choice:

- Generate a call via the target function's function descriptor, loading the entry point address and TOC pointer from the descriptor, or
- Generate a direct branch-and-link without loading a new TOC pointer.

The first course will always work (it's "safe") and, while not as efficient as the simple branch-and-link, is not excessively inefficient given that the compiler can schedule the call's instructions along with those for argument loading and other computation.

The second course is the default since it produces the most efficient linkage for what is expected to be the preponderance of cases. This is the same as is the default for AIX on RS/6000, and has the disadvantage that calls to functions in shared libraries are not resolvable at link-edit time. Instead, the linkage editor must resolve the call to "glue" code, a local sequence of instructions that implements the call by saving the caller's TOC pointer, loading the target's entry point address and TOC pointer, and issuing a branch via a register to the library routine.

Note that calls that are presumed at compile time to be local but that are discovered at link-edit time to be imported cannot possibly be as efficient as calls that are presumed at compile time to be imported. There is the overhead of branching to the glue code, the possibly redundant saving of the caller's TOC register (the linkage editor cannot know whether it is safe to omit this store[1]), and the fact that none of the glue code can be scheduled with the rest of the caller's code. This latter overhead is especially significant, as there are unavoidable dead cycles while the processor waits for the register to be loaded and the branch to the target to be resolved.

4.4 Making calls more efficient

With the proper use of #pragma statements (or via command-line arguments), the compiler can be induced to generate the most efficient linkage for each call.

No additional information need be provided for two categories of calls:

- Calls via function pointer. For these, the compiler always generates "imported" linkage.
- Calls to routines defined in the same source file as the caller. For these, the compiler can generate "local" linkage since they are by definition in the same executable as the caller. It is not incorrect for the compiler to generate "imported" linkage, but it is less efficient to do so.

This leaves calls by name to routines that are defined in compilations other than that of the caller. The compiler's default is to treat such a call as "local", which may entail call time overhead in the form of "glue" code. The following strategy when coding a source module should result in the most efficient linkage.

```
/*      A typical module, which calls
 *          -- system-supplied functions in various libraries, with
```

1. It has been proposed that a means be provided for the compiler to communicate to the linker that the store is not required in the linkage glue because it is redundant.

```
 *                   prototypes in system header files, and
 *             -- user-supplied functions linked together into a
 *                   single executable, with prototypes in local
 *                   header files
 */

#pragma  procimported

#include <stdio.h>
#include < . . . >              /* other system header files */

#pragma  proclocal

#include "localhdr.h"
#include " . . . "              /* other local header files */

main ()
{
   . . .
}
```

The intent here is that all functions with function prototypes in system header files are called via function descriptor, and all functions with function prototypes defined locally (in local headers or in the source program itself) are called via simple branch-and-link.

4.5 Consequences of "guessing wrong"

If the compiler is told via #pragma or command-line argument that a particular function foo() is local, and the linkage editor discovers at link time that foo() is in fact not part of the executable but is in a library, then glue code is inserted as described in the previous chapter. The resulting inefficiency can be eliminated by proper use of compiler directives.

Assuming that the linkage editor lists all the local calls that entail the introduction of glue code, this can be accomplished with reasonable effort. It is a simple matter to add the list of local call "failures" to a -qprocimported invocation parameter for the compiler, especially if a build tool such as make or nmake is being used. The source files are then recompiled, <u>without making any changes to the source files themselves</u>, and this time the link-edit should complete without glue code.

One would expect to treat critical applications in this manner, and hence it is a valuable function for the linker to produce the relevant information messages.

5 NT-Specific Conventions and Structures

The material in this chapter deals with items that are specific to the port of Windows-NT[1] to PowerPC. It should be read in the context of Chapters 1 - 4, for which it provides refinements and additional definitions.

5.1 Naming Conventions

5.1.1 Functions and function descriptors

As described in Section 3.5, two names are required: the name of the function descriptor and the name of the code entry point. For a function "foo",

- foo is the name associated with the descriptor
- ..foo is the name associated with the entry point of the procedure.

The unadorned name is the one that is used to reference a function — the address it denotes would be the value assigned to a function pointer, for example.

Except when writing assembler code, programmers generally need not be aware of the entry name for a function. The only name used in source code and imports and exports lists is the unadorned function name. The entry name is automatically generated by the compiler, together with the appropriate RLDs and symbol table entries to allow the linker to make the right connections. Entry names also appear in special stub sequences called glue code used in implementing calls to functions in shared libraries. These sequences are automatically introduced by the linker. Source language debuggers should allow users to use function names when setting entry point break points.

5.2 Stack frame layout

The stack frame layout in the NT implementation differs from that described in section 3.3.1. The stack frame header contains 6 words and the output argument area contains at least 8 words, providing the space to store the 8 general purpose parameter registers.

5.2.1 Stack frame header

Because the glue sequences in NT must save the link register, a slot is reserved for this purpose, and an additional slot is reserved for future use. The stack frame header in the NT implementation is

1. Windows-NT is a copywrite of the Microsoft Corporation

Offset (hex)	Contents
0	Back chain; points to caller's (next higher addressed) stack frame
4	Slot for glue code to save a register
8	Slot for glue code to save a register
C	Reserved for future use
10	(spare)
14	(spare)

Except for the back chain, a function *never* stores into any other words of its "own" stack frame header. Thus while a function is actively executing, only the back chain slot in its stack frame header contains useful data. This allows an efficient implementation of the alloca() function, which need only move the back chain field to a lower address in order quickly to allocate additional storage in the stack frame. Linker-inserted "glue" code (see Section 3.6.4) may use reserved locations in the header to save up to two registers, usually the link register and register 2. One location is reserved for future use. The remaining two words are "spares" that may be used by specialized utilities such as profilers, tracers, debuggers, etc..

5.2.2 Output argument area

This portion of the stack frame is used to contain arguments to be passed to called functions, and must be at least eight words long. See Section 5.4, "Parameter passing" on page 53, for details.

5.2.3 Minimum stack frame size

A program need not acquire a stack frame at all if it

- requires no stack storage outside the "slack space" (see Section 3.3.7), taking account of any other routines that may be using the slack space simultaneously, **and**
- calls no other functions that use the stack.

If a stack frame must be obtained, the minimum size frame consists of the header (6 words = 24 bytes) plus the smallest allowed output argument area (8 words = 32 bytes), for a total minimum size stack frame of 64 bytes since frame sizes must be a multiple of 16 bytes.

5.3 Calls to functions not in same executable

Section 3.6.4 discusses how linker introduced linkage glue is used to implement calls that are discovered, at link time, to be to functions not in the same executable. The NT implementation differs somewhat from that description because of differences in library management and linker operation. In addition, the glue sequences are different.

5.3.1 Functions in shared libraries

In the NT port, when a program is linked that involves procedures from a library, a .lib file is included as an argument to the link step. The .lib file contains, among other things, stub sequences of code (variously called "thunks") and some directory and symbol information pertaining to the actual code and its descriptor, which are contained in a corresponding .dll file. For example, if a procedure in the program calls a library function called foo, the .lib file will contain a sequence of instructions with name ..foo. Note that the name associated with the glue code is identical to the name associated with the code for the called function. Recalling that a local call to foo is implemented simply by a branch-and-link-relative to ..foo, the linker will automatically resolve the reference to the glue code in the .lib file with the same name. The form of the linkage glue is shown in Section 5.3.2. The NT linker does not construct it at link time; it merely copies it into the executable. The labels ..foo , foo.body, and foo.end indicate the three addresses needed for the function table entry associated with exception processing (see Section 5.7.7).

The .lib file contains sufficient information for the linker to construct the appropriate TOC entry for [toc]foo and the necessary relocation information to enable the later *load* step, in the presence of the .dll file, to result in correct references *to* the TOC and correct address information *in* the TOC.

5.3.2 Glue code sequences for NT

The glue code sequences used in NT save and restore the link register. This makes it unnecessary for language processors to reserve room, as indicated in Section 3.6.4, for an instruction that may be needed to restore the TOC register following a branch and link. The required load is included in the glue sequence.

For a call to an imported function foo, the glue sequence bound into the same module as its caller is:

```
..foo:
        lwz     r.11, [toc]foo(r.2)     # get address of foo's descriptor
        stw     r.31, glsave1(r.1)      # save r.31 preparatory to using it
        lwz     r.12, 0(r.11)           # get address of ..foo
        mflr    r.31                    # save link register in r.31
        mtctr   r.12                    # get ready to branch to ..foo
        stw     r.2, glsave2(r.1)       # save r.2
foo.body:
        lwz     r.2, 4(r.11)            # load foo's r.toc
        bctrl                           # branch and link to ..foo
  -----------
        mtlr    r.31                    # get ready to return
        lwz     r.31, glsave1(r.1)      # restore r.31
        lwz     r.2, glsave2(r.1)       # restore r.toc
        blr                             # return to caller
foo.end:
```

The labels foo.body and foo.end are included for clarity. They are not externally visible names and do not participate in name resolution. They denote the addresses that play a role in exception processing and stack unwinding. In particular, glue code must have entries in the tag table index (see Sections 3.9.2 and 5.7.7). This is a responsibility of the linker. The exception handler does not otherwise treat glue code in any special way.

5.4 Parameter passing

The following describes the parameter passing paradigm implemented for NT. Except for some minor details, it is the same as used in AIX and as described in the Power ABI document.

5.4.1 Location of arguments

For Little-Endian PowerPC, as for AIX on Power, we pass up to eight words in the general registers, loading them sequentially into general registers 3 through 10. In addition, up to thirteen floating point arguments can be passed in floating point registers 1 through 13. If fewer (or no) arguments are passed, the unneeded registers are not loaded and will contain undefined values on entry to the called function.

In only two cases, both relatively infrequent, must arguments be in storage for a call:

- When the amount of data being passed is more than will fit in the eight general registers (and the thirteen floating point registers) provided. The remainder is placed in storage.
- When the called function takes the address of one or more of its input parameters. For example, printf() does this when it interprets at run time the types of its parameters, and steps through them one at a time using the varargs macro.

To handle both situations, a fixed area in the stack frame is allocated to hold the data being passed. Figure 14 shows a detail of the **caller's** stack frame where space is reserved for arguments passed to a called function's parameters. The compiler reserves in the caller's stack frame enough space for the caller to construct the longest argument list necessary for all the functions that it calls. The eight shaded words are always reserved; these correspond to the eight general registers in which a precedure can find its parameters (registers 3 through 10). For any call requiring more than the eight registers, words after the eighth are stored in the caller's stack frame beginning with the slot labelled "Parameter word 8" in Figure 14. Nothing is stored in the eight shaded words before the call is made; only the overflow goes to storage.

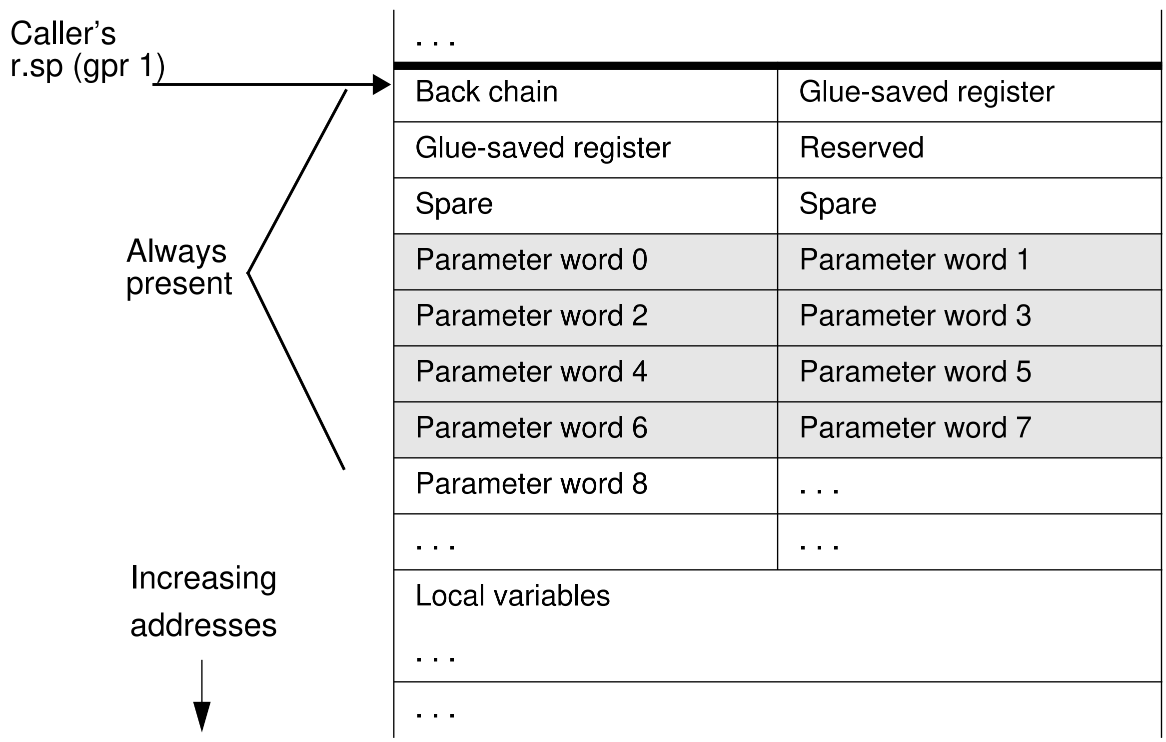

Figure 14 Parameter list area in stack frame

The called function accesses the first eight words of its parameters in general registers 3 through 10. For parameters beyond the eighth word, it must access the list constructed in the caller's stack frame. It is trivial to locate this: the compiler knows the length of the stack frame that it has built, and the list starts at a fixed location in the caller's stack frame as shown above. Thus the ninth parameter word is at offset 56, the tenth at offset 60, etc. Alternatively, since the caller's stack pointer is always stored at offset 0 from the current stack pointer, that value can be loaded and used to address the parameter list.

In the case where the called function takes the address of an input parameter, or treats its parameters as an array, the eight words allocated in the caller's stack frame can be used as targets of store instructions for registers 3 through 10. Since the eight words are always present, the called function can blindly store all eight registers without regard to their contents in order to materialze the argument list in contiguous storage.

5.4.2 Floating point arguments

Floating point arguments are passed in floating point registers 1 through 13. If there are more than thirteen such arguments, the remainder are passed in storage in the argument extension area, intermixed with any non-floating arguments as indicated in Section 5.4.3.

5.4.3 Mapping the argument list

The details of argument passing are most easily understood by considering first how the arguments would appear in storage if they were all passed that way. Then, the actual situation is derived from this conceptual argument list.

Assume that the conceptual argument list begins on a doubleword boundary. Now take all the arguments in order, including aggregate arguments passed by value (e.g., structures in C) and put them sequentially in the list, observing alignment requirements. For NT, these requirements are fullword for all arguments less than 8 bytes in length, and doubleword for all others. The result is the conceptual argument list, in which each argument appears exactly as it would appear in storage and each separate argument begins on *at least* a word boundary.

What is actually passed is like the conceptual argument list except that some of the arguments appear in registers rather than in storage. Exactly what happens depends on the existence and kind of a function prototype for the called function.

In the absence of a function prototype, the contents of the first eight fullwords (including any floating point or aggregate values) are placed in general purpose registers 3 through 10 rather than in storage. Additionally, the first thirteen floating point *scalar* arguments are placed in floating point registers 1 through 13 (despite the fact that they may already appear in a register or in storage). That part of the conceptual argument list containing values after the eighth fullword appears in the caller's stack frame beginning at offset 16 from the caller's stack pointer..

If a function prototype is present specifying all the parameters of the function[1], redundancy can be eliminated from the argument list: floating point values that appear in floating point registers need not also appear in general purpose registers or in storage. However, even in this case the space for them is retained. This is necessary because the called function must be able to access its parameters the same way whether or not a function prototype was available to its caller..

1. It contains no ellipsis, so no call instruction to the function supplies more arguments than there are parameters in the prototype. In effect, the function and its caller agree in advance about the number and types of parameters. This would not be true, for example, for a function like printf, which determines its parameter types at run time and uses the varargs macro or direct address arithmetic to access them.

If a function prototype is present and contains an ellipsis, arguments appearing before the ellipsis are treated as if a function prototype were present; all others are treated as if there were no function prototype.

5.4.4 Materializing the parameter list in storage

The parameter passing protocol described in section 5.4.3 permits a rather general implementation of functions that access parameters in a variety of ways by means of pointers. The key point is that there is sufficient information to permit the compiler to materialize the entire conceptual argument list in storage.

If a function takes the address of any of its parameters, it is the compiler's responsibility to ensure that the proper value exists in every part of the argument list that may be accessed via a pointer derived from that parameter address. This is guaranteed if the function's prologue contains code to store to their home locations all parameters that are passed in registers. Even if the number and types of the arguments that are passed to a function are not known at compile time, it is always safe to generate code to store the eight general purpose registers, and, following this, any floating point registers that are explicitly mentioned in parameter declarations.

Of course some optimization is possible. It is not really necessary to store a general purpose parameter register if a floating point registers will be stored to the same location. More signifi-canly, there are cases in which analysis by the compiler would indicate that not all of the argument list need be materialized; but in the absence of such a result, the address of any parameter must be treated as a potential alias of every part of the argument list, even parts of it that are not explicitly mentioned as parameters..

5.4.5 Return values

Certain values returned from a function are placed in parameter registers in the same way as they would be if they were being passed as arguments:: scalar non-floating data are returned in general purpose registers, and scalar floating values are returned in floating point registers. In particular, a single non-float scalar return value is returned in gpr3 and a float scalar value in fpr1. Languages that implement 64-bit integer data will pass such an item in two consecutive slots in the argument list, and will return it in gpr3 and gpr4.

A value that is not returned in a register (such as a structure in C) is written by the callee to a memory location provided by the caller The address of the location is passed as a (hidden) first argument in gpr3..

5.4.6 Some examples of parameter passing

Assume the following declarations:

```
typedef struct { int a,b,c; double dd;} sparm;
        /* assumed structure mapping requires dd to be doubleword aligned */
sparm s,t;
int x, *y;
char c;
double ff,gg;
```

Example 1:

```
x = noProto(x,c,y,ff,s,gg)
```

The indicated arguments would appear in the conceptual argument list, which begins at offset 24 (18 hex) from the caller's stack pointer, as follows: x at offset 24 (18 hex), c at offset 28 (1c hex), y at offset 32 (20 hex), etc. Because of alignment requirements, ff begins at offset 40 (28 hex), skipping the word at offset 36 (24 hex). Hence the actual places occupied by the arguments are as indicated in Figure 15.

gpr's		storage at offset		fpr's	
gpr3	x	18 (hex)			
gpr4	c	1c			
gpr5	y	20			
gpr6		24			
gpr7	ff(low)	28		fpr1	ff
gpr8	ff(hi)	2c			
gpr9	s.a	30			
gpr10	s.b	34			
		38	s.c		
		3c			
		40	s.dd(low)		
		44	s.dd(hi)		
		48	gg(low)	fpr2	gg
		4c	gg(hi)		

Figure 15 Parameter passing, example 1

Example 2.

```
sparm Proto(sparm,double, int,double);
t = Proto(s,ff,x,gg);
```

Since a structure value is being returned, the caller allocates temporary space for it in its stack frame and passes the address of the space as a (hidden) first argument.[1] The required alignment for structure s forces it to begin in register r.5 (associated with offset 32 = 20 hex) rather than r.4. The presence of the prototype makes it possible to pass the floating arguments in floating point

1. When possible, an optimizing compiler might be able to use the target of the assignment (t) directly as the temporary, thus avoiding a copy when the function returns.

registers only. Space for them must still be reserved, however, which explains why x appears at offset 64 (40 hex) rather than 56. The arguments thus actually appear as shown in Figure 16.

gpr's		storage at offset		fpr's	
gpr3	&temp	18 (hex)			
gpr4		1c			
gpr5	s.a	20			
gpr6	s.b	24			
gpr7	s.c	28			
gpr8		2c			
gpr9	s.dd(low)	30			
gpr10	s.dd(hi)	34			
		38		fpr1	ff
		3c			
		40	x		
		44			
		48		fpr2	gg
		4c			

Figure 16 Parameter passing, example 2

Example 3:

This example is rather simple and is included to show the basic efficiency of the parameter passing.

```
extern double foo(int *,double,int,int,double,double)
ff = foo(y,ff,x,*y,gg,gg+1.0);
```

y is in r.3, ff is in f.1, x is in r.7, *y is in r.8, gg is in f.2, and gg+1.0 is in f.3. The double is returned in f.1. The layout is shown in Figure 17. If the function prototype were not present, in addition to

gprs		offset from caller's stk ptr		fprs	
gpr3	y	18 hex			
gpr4		1C			
gpr5		20		fpr1	ff, return val
gpr6		24			
gpr7	x	28			
gpr8	*y	2C			
gpr9		30		fpr2	gg
gpr10		34			
		38		fpr3	gg+1.0
		3C			

Figure 17 Parameter passing, example 3

the above, the argument ff would also be in r.5 and r.6, gg would also be in r.9 and r.10, and gg+1.0 would also be in the double word of storage at offset 0x38 from the stack pointer. Notice that the presence of the prototype *allows* for a more economical passing strategy; it does not *force* it: no harm is done if the floating arguments appear in both places.

5.5 PE Module Format

Windows NT's object module structure is called Portable Executable, or PE. It is derived from UNIX[1] COFF, or Common Object File Format, with optional headers and additional sections as allowed for in the COFF architecture.

5.5.1 New COFF section types in PE

In order to support position-independent code via the TOC, and tag tables for exception handling and debugging, some new COFF section types are defined and special processing of TOC references is added. The strategy is as follows:

- Several new COFF section types are introduced:

.reldata	Like .data but intended to hold data subject to relocation, such as function descriptors.
.pdata	Tag index tables (*function table* for NT port) (for exception handling, debugging, etc.)

1. UNIX is a trademark of UNIX System Laboratories

.ydata Tag tables (*scope table* for NT port) (for exception handling, debugging, etc.)

.toc The TOC. This is actually a subsection of the .idata section.

- The new section types are generated by the assembler (and eventually directly by compilers) and are processed as are other distinct section types by the linker, up to the point where an executable module (".exe" file) is generated. When processing multiple input files, the PE linker concatenates together all the .text sections into a single .text section, the .data sections into a single .data section, etc. This same process is followed for the newly-added COFF section types for PowerPC: .reldata is concatenated with .reldata, .pdata with .pdata, and so on.
- When an executable file (".exe" or ".dll") is produced, the new sections cease to exist in their own right — their data are added to the data for the .data and .rdata sections.

  The read-write .data section is made up of the concatenation of:
  the TOC (see below, for NT this is part of the .idata section),
  the input .reldata sections, and
  the input .pdata sections, and
  the input .ydata sections.
  the input .data sections.

  The read-only .rdata section consists of the read-only (not to be modified after link time) program text.

  Grouping the TOC and other relocatable data together at the start of the read/write .data section minimizes the number of pages that must be modified when loading and relocating a module.

The new sections exist in the PE files only until they are bound into an executable module. They then become part of the ordinary read-write and read-only data sections and are not distinguished to the program loader, for example, as anything other than ordinary data. Thus only the assembler and linker need to be aware of the new section types.

5.5.2 Construction of the TOC

The information necessary to build the TOC is implicit in the symbol table and in the relocation information for instructions that refer to the TOC. There is no separate TOC COFF section in ".obj" files. The symbol and relocation information is maintained by the binder until an executable file is finally produced. At this point the entries that make up the TOC are materialized in the .idata section..

For example, assume that two different object files contain code that references an external variable x. There is a symbol table entry for x and several relocation entries that refer to "TOC entry for x", but no instantiation in the ".obj" files of an actual pointer to x. From the existence of the "TOC entry for x" relocation entries, the PE linker will infer that it must add such an entry to the TOC constructed for the ".exe" or ".dll" file. It will then cause all the references to "TOC entry for x" to refer to the single entry in the TOC.

Provision is made in the assembler for references to four kinds of TOC entries:

- [toc] x refers to a default-width TOC entry (currently 32 bits) pointing to x.
- [toc32] x refers to a 32-bit-wide TOC entry pointing to x[1].
- [toc64] x refers to a 64-bit-wide TOC entry pointing to x[2].

- [tocv] x refers to an entry in the TOC that contains x.

For the first three, the assembler will produce RLDs that cause the linker to introduce cells into the TOC. The last will cause an external reference to a cell that the linker is expected to include in the TOC.

5.6 Linking programs

Since one of the concerns of the NT port were to minimize changes to existing code, some of the features of linking work differently than they do in AIX and Power. The following describes, in a general way, some of the concepts.

5.6.1 Imported items

When linking a module (executable or DLL), symbols that are not resolved locally may be resolved to items exported from a DLL. To be resolved, such items must be represented by an entry in the .idata section of a .lib file that is listed as one of the inputs to the link command. The linker will incorporate such .idata entries into the TOC of the module.

Following the NT linker semantics, the symbol associated with the .idata entry for an exported data item is given the name of the item, even though it actually represents the address of the item. Thus, programmers will have to be aware of whether a data item is imported or not when they write their code. References to an imported item can only be made by a pointer variable that is given the same name as the exported data item. For example, if confusion is the name of an exported integer, then the programmer that wishes to access confusion must code

```
extern int   * confusion;
```

which will result in resolving his *pointer* variable confusion to the .idata cell containing the address of the exported *integer* confusion. As was mentioned, this cell ends up in the module's TOC.

The situation is a little different for exported functions. In that case, whether or not a function is resolved locally need not be known in advance. When a function, say foo, is exported, the entry in the .idata section of the appropriate .lib file is associated with the name __imp_foo. In addition, the .lib contains a glue code sequence named foo.ep and a function descriptor named foo. The glue code sequence is:

```
foo.ep:
        lwz      r.11, [tocv]__imp_foo_ep    #get address of function descriptor
        stw      r.31, glsave1(r.1)          #save r.31 preparatory to using it
        lwz      r.12, 0(r.11)               #get address of the real foo.ep
        mflr     r.31                        #save link register in r.31
        mtctr    r.12                        #get ready to branch to foo's code
foo.body:
        lwz      r.2, 4(r.11)                #load foo's TOC address
        bctrl                                #branch and link to the real foo.ep
----------------
        mtlr     r.31
        lwz      r.31, glsave1(r.1)          #restore r.31
```

1. Not supported in all NT assemblers.
2. Not supported in all NT assemblers.

```
lwz    r.2, glsave2(r.2)    #restore r.toc
blr                         #return to caller
```

The .idata entry for __imp_foo points to the "true" function descriptor for foo in the exporting DLL, and ends up in the local TOC. As expected, a call to foo is implemented by a local bl to foo.ep, the glue sequence.

This arrangement supports, albeit somewhat awkwardly, static initialization of function pointer variables to exported functions. For example, the variable pf in

```
void static (*pf)() = foo;
```

would be initialized to the address of the descriptor foo associated with the glue sequence indicated above. The awkwardness comes about because calls to an imported function via a statically initialized function pointer now traverse two glue sequences: one in line and the other introduced during the link step[1]. An additional awkwardness is due to the fact that the address of a function is only unique within a module. Thus, the ANSI semantics of pointer function comparison are not fully satisfied.

5.6.2 Relocation entries ("RLDs")

The standard PE file definition includes a structure to hold relocation information for use by the linker and the loader. For Little-Endian PowerPC, the structure is defined like this:

```
typedef struct _IMAGE_RELOCATION {
        ULONG  VirtualAddress;
        ULONG  SymbolTableIndex;
        USHORT Type;
} IMAGE_RELOCATION;
```

The value contained in the 16-bit "Type" field is completely machine-dependent. A scan of the definitions for various machines contained in the ntimage.h header file reveals that there is almost no overlap in relocation type values from one machine to the next, so the values chosen for Little-Endian PowerPC do not have to fit into an existing grand scheme. We define the 16-bit type field as shown in Figure 18.

- ADDR64, ADDR32, and ADDR16 types are used for fields that are to hold addresses; these can be data fields (doublewords, words, halfwords) and, in the case of ADDR16, the 16-bit displacement field in a load or store instruction.
- ADDR24 and ADDR14 types are used for addresses in branch and conditional branch instructions. As indicated in the instruction format diagrams in the PowerPC Architecture books, these represent 26-bit and 16-bit addresses, respectively, whose two low order bits are presumed to be zero; the bits in the instruction format that would normally be occupied by these two low order bits are used for other purposes and must be preserved when relocation is performed on the instruction.
- REL24 and REL14 types are used for branches relative to the IAR. They represent 26-bit and 16-bit displacements; the same comments as above for ADDR24 and ADDR14 apply here.
- TOCREL16 type is used for 16-bit displacement fields in instructions that reference cells in the TOC, such as lwz. The TOCREL14 type is used for 16-bit displacement fields

1. Linker modifications to avoid this problem are under consideration.

8 bits | 8 bits
Flags | Reloc

Type field in relocation entry

Name	Value	Description
NOOP	0x00	
ADDR64	0x01	64-bit address
ADDR32	0x02	32-bit address
ADDR24	0x03	26-bit address, shifted left 2 bits (e.g., branch absolute)
ADDR16	0x04	16-bit address
ADDR14	0x05	16-bit address, 2 low-order bits removed (e.g., load doubleword)
REL24	0x06	26-bit IAR-relative offset, 2 low-order bits removed (e.g., branch relative)
REL14	0x07	16-bit IAR-relative offset, 2 low-order bits removed (e.g., branch conditional relative)
TOCREL16	0x08	16-bit offset from TOC base
TOCREL14	0x09	16-bit offset from TOC base, 2 low-order bits removed (e.g., load doubleword)
ADDR32NB	0x0A	32-bit offset w/o image base
SECREL	0x0B	offset relative to containing section
SECTION	0x0C	section header number

Name	Value	Description
NEG	0x01	subtract relocation value rather than adding it
BRTAKEN	0x02	set branch prediction bit to predict branch taken
BRNTAKEN	0x04	set branch prediction bit to predict branch not taken
TOCDEFN	0x08	map cell for symbol in the TOC

Figure 18 Relocation entry type field

whose two low-order bits are presumed to be zero (similar to ADDR14 and REL14), and is used for certain load instructions such as ld.

- ADDR32NB Address relative to the virtual origin.
- SECREL is used to indicate that the value should be relative to the beginning of the section that contains the symbol.
- SECTION For fast access ot the header of the section containing the item. This is used by the *Codeview* debugger.

Each of the above values can be combined with one or more flag bits:

- NEG indicates that the relocation value is to be subtracted from the field in storage rather than added to it.
- BRTAKEN and BRNTAKEN indicate that the branch prediction bit (the low-order bit of the BO field) in a conditional branch instruction should be set to predict branch taken or branch not taken. Which way the bit should be set depends on the "sign bit" of the branch target address or offset. This is not in general known until link-edit time,[1] which is why there are relocation flags to indicate branch prediction.

  The assembler or compiler should set BRTAKEN or BRNTAKEN only on conditional branches and then only such branches as are specified by the programmer (in assembler language) or deduced by the compiler to require other than the default prediction. If neither flag is set, the branch prediction bit in the instruction should not be altered when the instruction is relocated.
- TOCDEFN indicates that no new TOC cell should be made by the linker. Instead, the linker should expect that the data corresponding to the symbol have been mapped into the TOC. The value associated with the symbol is the location of the data.

5.7 Exception processing

The model of program exception processing in Windows-NT[2] is implemented on Little-Endian PowerPC. Tag table indexes and tag tables follow the format indicated below. In addition, certain restrictions apply to function prologues to support the *reverse execution* mechanism that is implemented for stack unwinding.

5.7.1 Function table (Tag table index) format

For the NT port, the tag table index is called the ***function table***, and the format of its entries is defined by:

```
typedef struct _RUNTIME_FUNCTION {
        unsigned long BeginAddress;
        unsigned long EndAddress;
        PEXCEPTION_ROUTINE ExceptionHandler;
```

1. In principle, the way that the branch prediction bit should be set for branch absolute may not be known until program load time. PowerPC object code may be considered **not** to be position independent because of this. However, in a real-world system we can take the position that all the functions reached via branch absolute will be in low storage (or conversely that all such code will be in high storage) and thus set the branch prediction bit properly at link-edit time.
2. Copyright Microsoft Corp.

```
        void * HandlerData;
        unsigned long PrologEndAddress;
} RUNTUME_FUNCTION, *PRUNTIME_FUNCTION;
```

The significance of most of the fields is obvious from their names. ExceptionHandler (if not null) is a function pointer to a language specific handler that knows how to process the exception constructs of a particular language, and HandlerData (if not null) points to the scope table that contains data used by ExceptionHandler.

In general, every procedure that modifies r.sp (i.e., acquires a stack frame) or the link register or any non-volatile register must have an entry in the function table. As an example, for a C procedure with entry name pooh, the assembler instructions for producing function table entry definitions are:

```
.pdata
.align    2
.globl    pooh.ep                        # only if pooh is not static
.globl    __chandler
.long     pooh.ep, pooh.end, __chandler, pooh.cdata, pooh.body
```

The ExceptionHandler entry containing __chandler describes the C language specific exception handler, and the HandlerData field containing pooh.cdata points to the data used by that handler. If there are no try blocks, the ExceptionHandler entry should be null. For programs written in other languages, the handler and the data it uses may be different or even absent. In the latter case, the ExceptionHandler field should again be null.

Note that only pooh.ep and __chandler are global symbols. pooh.end indicates the end of the function's code, and pooh.body indicates the end of its prologue.

If the ExceptionHandler field is null, then the HandlerData field may either be null or contain a specific code identifying some property of the code for which this is an entry. Specific codes and their meanings are listed below.

Code	Significance
0x00	None
0x01	Register save millicode
0x02	Register restore millicode
0x03	Glue code sequence

Figure 19 Special codes in HandlerData field when ExceptionHandler = null

5.7.2 Scope table (C-specific tag table) format

For the NT port, the tag table associated with a C program is called the ***scope table***, defined as:

```
typedef struct _SCOPE_TABLE {
        unsigned long Count;
        struct
        {
            unsigned long BeginAddress;
            unsigned long EndAddress;
```

```
                unsigned long HandlerAddress;
                unsigned long JumpTarget;
            } ScopeRecord[1];
    } SCOPE_TABLE, *PSCOPE_TABLE;
```

BeginAddress and EndAddress are the addresses that delimit the code for a *try* block; HandlerAddress is the address of the code for the exception filter (for a *try-except*) and for the termination handler (for a *try-finally*); and JumpTarget is the address of the handler code for a *try-except* and is 0 otherwise. The scope table entries are sorted primarily by EndAddress and secondarily inversely by BeginAddress, so innermost *try* blocks appear first. Exception filters and termination handlers are implemented as if they were scoped procedures statically nested in the procedure (or block) that contains them.

Scope tables are aggregated by the linker into the .ydata section. Appropriate assembler syntax for producing a scope table entry describing a try-finally block within the try part of a try-except block is

```
    .ydata
pooh.scopetable:
    .long 2
    .long tryf.begin, tryf.end, term.ep, 0
    .long trye.begin, trye.end, filter.ep, ehandler
```

where the symbols are local labels in the code that identify the various parts.

5.7.3 Up-level addressing

The NT implementation of termination handlers and exception filters treats these as statically nested functions such as occur in languages like algol and pascal. Since these functions do not share their stack frame with their parent, some means must be provided for them to address variables located in their parents' stack frames, i.e., their *environment*. The technique used is known as *static back chains*. Instead of the TOC address, the environment address is passed to statically nested blocks.

Care must be taken in the compiler to ensure that the environment of any block that contains a nested block is stored in the stack frame of the containing block so that the nested block can access it. One may consider the environment of a level-1 block (the outer function) to be its TOC. Thus, each nested block can access all of its inherited variables by following the chain of environment pointers: the *static back chain*.

In the NT implementation, the address of a nested block's environment is defined to be the stack frame pointer of the function that called, or *established*, the environment. If foo called bar, and bar contained a termination handler, then the handler's environment is foo's stack pointer: the *establisher frame* for bar.

5.7.4 Stack Unwinding

Simply stated, *unwinding a frame* is the process that, given an instruction address (pc value) and a state, produces the instruction address of the branch instruction that implemented the call to the function containing the given instruction, and produces the state at the time of the branch. The state in this context means the values of all the non-volatile registers. For languages whose run time environment includes a dynamic stack, the state also includes the relevant address (sp) of the stack frame active when an instruction is executed. Although reasonably general, the following discussion should be understood to be primarily focussed on the NT port.

Two basic steps are involved in stack unwinding:

- Given a pc (value), identify the procedure containing the instruction
- Knowing the procedure, restore the state to what it was when the procedure was called.

The first step makes use of the function table. A binary search will determine whether there is an entry for the procedure. If there is no entry, then (barring any special handling) the state is presumed to be unchanged from when the instant procedure was called, and the pc of the branch instruction is presumed to be in the link register.

If there is an entry, then the entry indicates the address of the last instruction in the procedure's prologue. Assuming that the current pc is past the end of the prologue, simulating the reverse execution of a properly constructed prologue will restore the state of the machine to what it was when the procedure was called. That includes the value in the link register, which then gives the pc for the calling *branch-and-link* instruction. If the pc is *not* past the prologue, then a simulated reverse execution of that part of the prologue up to the pc will have the same effect.

It is clear from the above that the two major restrictions on procedure formation that are imposed are that

- Prologues occur only at the beginning of procedures and their last instruction is unambiguously identifiable, and
- Given a state and an actively executing instruction *anywhere* in a procedure, simulated reverse execution of the procedure's prologue (or part preceeding the instruction) must be able to restore the state (non-volatile registers and program counter) to what it was at the time the procedure was called.

5.7.5 Prologues and stack unwinding

If "reverse execution" of a function's prologue is a part of stack unwinding, that function must have an entry in the function table. The entry specifies the first instruction (beginning of the procedure) and the byte following the last instruction of its prologue. This satisfies the first of the two restrictions on prologue formation mentioned in the previous section.

A definition of what a prologue can consist of to achieve the second restriction depends on how complicated a stack unwinder one is willing to write. For the NT port, it seems desirable to specify fairly constrained prologues, leaving for the future increased generality based on sophisticated enhancements to the unwinder(s). The following defines a class of prologues sufficient to achieve the current goals. The idea is that only certain instructions will be recognized by the stack unwinder to be executed "in reverse". Any other instructions that are encountered are simply ignored.

5.7.5.1 Recognized prologue instructions

The following instructions are recognized by the stack unwinder to be a part of a legitimate prologue:

mflr	r_x
mfcr	r_x
lwz[1]	r_x,k(r.1)# where k = 0 or 4 or 8 or 12
stw	r_x, disp(r.1)
stfd	f_x, disp(r.1)

1. This instruction is used to move a stack frame header and should only occur in code that does not always allocate 32 bytes in the caller to hold the values in the eight parameter gprs.

```
mr      rx, ry
bl(a)   millicode_to_save_registers
stwu    r.1, –stack_size(r.1)
stwux   r.1, r.1, rx
addi    r.12, r.1, N
```

where rx and r_y are any arbitrary registers and disp, N, and -stack_size are legitimate values for the displacement field of an instruction. The bl instructions are only recognized if their targets, as determined from the function table, are some relevant millicode. Where specific registers are mentioned, only instructions with those registers in the indicated positions are recognized.

Although other instructions can appear intermixed with prologue instructions, they will be ignored by the stack unwinder. Recognized instructions that are NOT a part of the prologue may only be intermixed if they result in no net effect on stack unwinding. The safest way to form a prologue clearly is to include only prologue instructions, but allowing other instructions to be included may sometimes offer a scheduling advantage.

5.7.5.2 Prologue formation guidelines

- Use only recognized instructions to save state. This means that only r.sp (general register 1) should be used as a base register to save non-volatile registers in storage by in-line code.
- Stack frame acquisition should always be done using either the stwu or the stwux instruction, and these are the only instructions that should modify r.sp.
- The only instruction to be used to calculate values into r.12 for use in the millicode to save registers is either mr r.12, r.1 or addi r.12, r.1, N.
- All non-volatile registers that are changed in the procedure must be saved in the prologue. This includes the link, TOC, and (non-volatile part of the) condition register.
- Non-prologue instructions that look like recognized instructions may be intermixed with prologue instructions only if their reverse execution is harmless. Non-recognized instructions will be ignored by the unwinder.
- The byte following the last instruction in the prologue must be indentified in the *function table* entry.

5.7.5.3 Reverse Execution

The following brief description of how the unwinder does reverse execution is presented to provide some guidance for prologue formation.

Starting from the last instruction in the prologue (or the instruction immediately preceeding the current instruction if it is at a lower address), the stack unwinder works its way backwards through instructions, simulating a "reverse execution" of instructions that it recognizes as follows:

mflr		*reversed as*	mtlr	r_x	
mfcr	r_x	*reversed as*	mtcrf	255, r_x	
lwz	r_x,k(r.1)	*reversed as*	stw	rx,k(r.1)	k=0,4,8,or 12
stw	r_x, disp(r.1)	*reversed as*	lwz	r_x, disp(r.1)	
stfd	f_x, disp(r.1)	*reversed as*	lfd	r_x, disp(r.1)	
mr	r_x, r_y	*reversed as*	mr	r_y, r_x	*if r_y is not r.1*
mr	r.12,r.1	*is reversed only as a part of reversing bl save_registers*			
addi	r.12,r.1,N	*is reversed only as a part of reversing bl save_registers*			

```
stwu    r.1, N(r.1)       reversed as            lwz   r.1, 0(r.1)
stwux   r.1,r.1,rx        reversed as            lwz   r.1, 0(r.1)
bl  save_registers        reversed as follows:
        First establish the value of r.12 on entry to the millicode sequence: search for
        the closest preceeding instruction that computes r.12: call it "E".
        If no intervening instruction changes r.1, then
                          compute                E.
        Otherwise, first compute                 lwz   r.12, 0 (r.1)
                          Then, if E is  addi r.12,r.1,N
                          then compute            addi r.12, r.12, N
                          Otherwise              ignore E.
        Finally,          simuoate the reverse execution of that instruction sequence from
                           the last instruction to the point where the sequence was entered.
```

All other instructions are ignored.

Note that if it is necessary to compute a value into r.12 to be used by millicode, the computation must be done by a single instruction. Other restrictions on prologue formation follow obviously from the indicated actions taken by the unwinder: a register that is to be preserved cannot be destroyed before it is saved, and all saving must be accomplished by the end of the prologue.

5.7.5.4 Reverse execution examples

The following is an example of a legitimate prologue:

```
begin:  mflr    r.0
        stw     r.31, -4(r.1)
        stwu    r.1,-64(r.1)
        add     r.3, r.3, r.4
        stw     r.0, 56(r.1)
body:
```

The label "begin:" marks the beginning of the procedure containing this prologue, and the label "body:" marks the end of the prologue. The add instruction is not part of the prologue code but was scheduled in the delay slots of the move-from-link-register instruction. Unwinding would ignore it and simulate the following instructions:

```
        lwz     r.0, 56(r.1)
        lwz     r.1, 0(r.1)
        lwz     r.31, -4(r.1)
        mtlr    r.0
```

A more complicated prologue example is:

```
begin:  mflr    r.0
        addi    r.12, r.1, -80
        bl      ._savefpr_22
        bl      ._savegpr_26        # known to use r.12
        stw     r.0, -108(r.1)
        stwu    r.1, -512(r.1)
body:
```

Here the stack unwinder does the following:

```
        lwz    r.1, 0(r.1)
        lwz    r.0, -108(r.1)
        addi   r.12, r.1, -80         # the closest preceding instruction for r.12
        bl     ._restgpr_26
        bl     ._restfpr_22
        mtlr   r.0
```

A similar prologue shows how to deal with intervening changes to r.1:

```
begin:  mflr   r.0
        bl     ._savefpr_22
        addi   r.12, r.1, -80
        stwu   r.1, -512(r.1)
        stw    r.0, -108+512(r.1)
        bl     ._savegpr_26
body:
```

Stack unwinding would look like

```
        lwz    r.12, 0(r.1)           # an intervening instruction changed r.1
        addi   r.12, r.12, -80
        bl     ._restgpr_26
        lwz    r.0, -108+512(r.1)
        lwz    r.1, 0(r.1)
        bl     ._restfpr_22
        mtlr   r.0
```

The next example indicates how a very large stack frame (more than 2^{15} bytes) *might* be handled:

```
begin:  mflr   r.0                    #prepare to save link register
        addis  r.12, 0, high_part
        addi   r.12, r.12, low_part   # compute negative of stack size into r.12
        stw    r.0, link_save(r.1)    # save the link register
        bl     _RtlCheckStk.12.ep     # run time routine that guarantees OK stack
        stwux  r.1, r.1, r.12         # resets r.1 and stores into back chain field
body:
```

Unwinding effectively does

```
        lwz    r.1, 0(r.1)
        lwz    r.0, link_save(r.1)
        mtlr   r.0
```

5.7.6 Epilogues

In order to perform unwinding correctly, it is necessary to know whether the current values of the pc and r.sp are *consistent*. In this context, consistent means that, whenever the instruction at pc is part of a procedure that "buys" a stack frame, r.sp correctly identifies that frame. That would not be the case, for example, at the instruction marked by * in the following code, a typical part of an epilogue

```
        addi   r.1, r.1, 248
*       blr
```

Here the stack pointer has been restored to the caller's frame but the instruction is still in the called procedure. In order to handle this situation correctly, it is necessary to be able to recognize such a situation. This is possible if the following restriction on epilogue formation is observed:

- Code to restore the stack pointer must be a *single* instruction that changes r.sp and occurs immediately preceding a return blr or immediately preceding a branch absolute to register restore millicode (which ends by implementing the return).

The above sample is legitimate. Another example is:

```
lwz     r.12, 0(r.1)
addi    r.12, r.12, -32        # point to end of gpr save area
lwz     r.0, -56(r.12)         # fetch return address
bla     ._restgpr_27
mtlr    r.0
addi    r.1, r.12, 32          # restore stack pointer
ba      ._restfpr_24
```

An illegal epilogue example is:

```
lwz     r.1, 0(r.1)
lwz     r.0, -8(r.1)
mtlr    r.0
lwz     r.31, -4(r.1)
blr
```

It is illegal because the instruction restoring the stack frame is not adjacent to the return branch.

This restriction allows the stack unwinder to determine the correct pc at whatever instruction the exception occurs. When it is a blr, the target of the blr is the appropriate pc: i.e., the instruction at that location is consistent with the value of the stack pointer. When it is a ba to register restore millicode, then again the value in the link register is the appropriate pc. In this case, the first step in the "unwind" is to simulate the execution of the restore millicode. For all other instructions, the pc and r.sp are immediately consistent.

Notice that this algorithm gives the correct results even if an exception occurs at a blr that does not exit the procedure.

Constructing a consistent pc and r.sp by trying to recognize the stack frame resetting instruction immediately prior to an exiting branch and "reversing" it is not sufficiently robust: that instruction may not be complete. It could, for example, simply be an mr or the second of an addis-addi pair. Indeed, if the procedure in question does not have its own stack frame, there will not even be a stack frame resetting instruction.

5.7.7 Function tables for linkage glue

Linkage glue is not treated in any special way by the exception processor since it is not generally possible to recognize it. Therefore, a function table entry should be made for each separate glue code sequence. The appropriate assembler instructions corresponding to the linkage glue shown in Section 5.3.2 are:

```
.pdata
.long   ..foo,foo.end,0,3,foo.body
```

which correctly identifies the boundaries of the glue and its prologue. No stack frame is associated with glue, so no exception handler need be specified.

5.7.8 Millicode

Register save/restore millicode sequences are recognized by the exception handler from the function table entries. For register save millicode, the entry is

```
address of first instruction in register save millicode
address of last instruction in register save millicode
0x0
0x1
address of first instruction in register save millicode # i.e., no prologue
```

For register restore millicode, the entry is

```
address of first instruction in register restore millicode
address of last instruction in register restore millicode
0x0
0x2
address of first instruction in register restore millicode # i.e., no prologue
```

5.7.8.1 Recognized register save/restore millicode instructions

Register save/restore millicode is not limited to the sequences shown in Section 3.6.8.1. Additional, unpublished sequences may be formed. The following instructions are recognized by the stack unwinder as register save millicode instructions:

```
mfcr   r.x
stw    r.x, disp(r.y)    where r.y is r.1 or r.12
stfd   f.x, disp(r.y)    where r.y is r.1 or r.12
mr     r.x, r.y          if r.y is not r.1
stwu   r.1, -stack_size(r.1)
stwux  r.1, r.1, r.x
```

The final instruction in reister save millicode must be blr. All other instructions are prohibited in register save millicode, including these instructions, which are recognized by the stack unwinder as prologue instructions

```
mflr   r.x
addi   r.12, r.1, N
```

The following instructions are recognized by the stack unwinder as register restore millicode instructions:

```
mtlr   r.x
mtcrf  255, r.x
lwz    r.x, disp(r.y)
lfd    f.x, disp(r.y)
mr     rx, ry
addi   r.x, r.y, N
```

The final instruction in register restore millicode shall be blr. All other instructions are prohibited in register restore millicode.

5.7.8.2 Millicode and stack unwinding

If an exception occurs at a bl instruction to one of the register save entry points, no special processing takes place. The stack unwinder starts simulating the reverse execution with the preceeding instruction. If an exception occurs at an instruction *in* register save millicode, no special processing takes place. The function table entry indicates that the function has no prologue and the stack unwinder continues with the instruction (presumably the bl to the millicode) with no change to the stack pointer. The stack unwinder starts simulating the reverse execution of the prologue in this frame with the instruction preceeding the bl. If an exception occurs at a b instruction to one of the register restore entry points, the stack unwinder must simulate the forward execution of the register restore millicode. If an exception occurs at an instruction in register restore millicode, the stack unwinder must simulate the forward execution of that instruction and the remaining instructions in the register restore millicode.

5.7.9 Considerations for language processors

It is expected that language processors will treat the try-except and try-finally functions consistently with the mechanism described. That means that exception filters and termination handlers will be implemented as out of line "internal" procedures, expecting to find in r.toc the address of their *environment*, i.e., the stack pointer (actually the frame pointer, which may differ from the stack pointer if an alloca() has been executed) of the function that contains them. Since the TOC address, or other base address for accessing static data, is treated as a parameter of an external procedure, it is handled like any other variable in the environment of an internal procedure. This means that, should the value be required in the internal procedure, it must be "driven home" to storage before the internal procedure is invoked. For TRY blocks, the call to the filter or termination handlers may not be explicit, so care must be taken to ensure that the value is in storage whenever such a procedure might be called. Parameters passed to the filters and termination handlers are as defined by MS (TBD).

The dispatcher will effectively branch directly to the exception handler. If it is found to be convenient, language processors may choose to implement exception handlers as internal procedures. In that case, they should specify the address of a call instruction to "their" handler as the location to which the exception dispatcher should branch. Optimizers must be careful to avoid introducing dependencies on values that may not be available when control is transferred to an exception handler. As mentioned above, the control flow to an exception filter, termination handler, or exception handler is not in general explicit, so care must be taken to guarantee that any values that are expected by these code segments to be in storage are in fact there. It is recognized that these constraints may impede some optimizations, but that is a consequence of the semantics of this exception mechanism.

Finally, language processors are expected to generate properly formed function table entries and scope tables.

5.8 Linking to millicode on Windows NT

Millicode sequences have well-known entry names that participate in name resolution at link time. These routines should not be exported from a library because they cannot be called through linkage glue. Therefore, they are bound together in the same executable with their callers.

6 ELF Module Structure for WorkPlace

This is described in a separate document: Karhi, K., *System V Application Binary Interface, PowerPC Processor Supplement (Draft)*, Ver. 0.7, October 27, 1993.

7 Assembly language

This section covers the assembly language statements necessary to build the code and data structures described in this document. Minor syntax differences may exist between different assemblers and what is described below. The reference manuals for a particular assembler should be consulted for complete and accurate information.

7.1 Pseudo-ops

The following pseudo-ops define COFF sections:

Pseudo-op	Read/ write	Description
.text	RO	Program text (instructions)
.data	RW	Initialized data (read/write)
.rdata	RO	Initialized data (read-only)
.bss	RW	Uninitialized data
.reldata	RW	Initialized data (read/write), with relocation
.pdata	RO	Tag index tables (for structured exception handling)
.ydata	RO	Tag tables (for structured exception handling)
.section	(various)	Can be used to define named sections with attributes as specified in a string parameter

Each of these pseudo-ops starts a COFF section in the assembly, or continues one of the same type started earlier. In each compilation there is no more than one section of each of these types. The section pseudo-ops take neither labels nor operands.

7.2 Writing a TOC reference

The following instruction loads into register 3 the entry from the TOC that points to foo, using register 2 as the base address of the TOC:

```
lwz     r.3, [toc] foo (r.2)
```

The construct "[toc] foo" means "offset of TOC entry pointing to foo". When combined with the base register (rtoc) it forms a "displacement(base)" address, which is the form of address valid for "lwz" (load word and zero).

7.3 Putting data into the TOC

The assembler can generate information that will cause the linker to map data into the TOC. Suppose that omelet is the name of a global 12-byte item to be mapped into the TOC and initialixed to all 0-bits. In the program containing the defining instance, the following would be coded:

```
        .globl   omelet
        .toc                  # keyword denoting the TOC subsection of .data
omelet:
        .long    0,0,0
```

In a program that merely referenced data but expected to find it in the TOC, storage references to the item that occurred in the code would appear, for example, as

```
        stw      r.5, [tocv]omelet (r.toc)
```

causing the assembler to produce an external reference with the TOC attribute in the symbol table, and also setting up a tocrel RLD with the DATAINTOC flag on.

7.4 Built-in symbols

Certain symbols are pre-defined by the assembler to stand for register numbers, condition register field numbers, well-known addresses, and the like. See Figure 20. All the symbols have a period (.) in them to avoid possible conflict with programmer-defined names in C programs.

Symbol	Definition	Description
r.0 . . . r.31	0 . . . 31	General register 0 . . . General register 31
r.sp	1	Stack pointer (general register 1)
r.toc	2	TOC pointer at entry (general register 2)
r.env	11	Environment pointer (general register 11)
f.0 . . . f.31	0 . . . 31	Floating point register 0 . . . Floating point register 31
cr.0 . . . cr.7	0 . . . 7	Condition register field 0 . . . Condition register field 7
.toc	—	Base address of the TOC, for use in function descriptors.

Figure 20 Predefined symbols

Symbol	Definition	Description
.textbegin	—	First byte of text section (code) in the executable module.
.databegin	—	First byte of data section in the executable module.

Figure 20 Predefined symbols

7.5 Function descriptors

A function descriptor consists of two words: the address of the function's code and the address of its TOC. For efficiency at link and load time, function descriptors should be placed in one of the sections containing information that is subject to relocation: the .reldata (relocatable read/write data) section or the section containing the TOC. To write a function descriptor for a function named "bletch", one could code

```
        .reldata
        .globl      bletch, bletch.ep       # make descriptor, code names visible
bletch:                                     # label on descriptor
        .long       bletch.ep, .toc         # address of code, address of TOC
```

A function descriptor should be supplied for every function or procedure, with one exception: a descriptor is not needed for a function declared static **and** whose address is not taken (it is not incorrect to generate a function descriptor even in that case).

For static functions, neither the name of the descriptor (bletch) nor the name of the code (.bletch) should be made globally visible. For all other functions, both names should appear as operands on .globl statements.

If one wanted to map a copy of the descriptor into the TOC, one would replace the .reldata keyword with .section,"t". Note that it is somewhat dangerous to make copies of function descriptors if the addresses of such copies are used as function pointer values. Since there may be several different copies, the semantics of function pointer comparison may be violated.

7.6 A "compilation" example

As an example of the assembly language needed for a complete program, consider the simple C program shown below.

```
extern int exint;              /* externally-defined int */
int inint1;                    /* globally visible int defined here, uninitialized */
int inint2 = 1;                /* globally visible int defined here, initialized */
static int stint;              /* locally visible int */
#pragma procimported exproc  /* specify imported linkage */
extern int exproc (int);       /* externally-defined function */
int inproc (int);              /* globally visible function defined here */
static int stproc (int);       /* locally visible function */

int main ()
{
    int rc;

```

```
14          exint = 2;
15          inint1 = 3;
16          stint = 4;
17
18          rc = exproc (5);
19
20          rc = stproc (6);
21
22          return rc;
23      }
24
25      int inproc (int x)
26      {
27          return (x + 1);
28      }
29
30      static int stproc (int y)
31      {
32          return (y + 2);
33      }
```

A correct, if not very efficient, assembler language equivalent to this program is:

```
# line 1
        .extern   exint                 # externally defined

# line 2
        .data                           # switch to ".data" section
        .globl    inint1                # make label visible globally
        .align    2                     # force to word alignment
inint1:
        .space    4                     # reserve 4 bytes, uninitialized

# line 3
        .globl    inint2                # make label visible globally
inint2:                                 # already on word boundary
        .long     1                     # word initialized to '1'

# line 4
stint:                                  # already on word boundary
        .space    4                     # reserve 4 bytes, uninitialized

# line 6
        .extern   exproc                # defined externally

# line 7
        .globl    inproc                # make label on fndesc visible globally

# line 10
        .reldata                        # switch to "relocatable data" section
```

```
            .globl   main                         # make label on fndesc visible globally
main:                                             # fndesc name = C function name
            .long    .main, .toc                  # assemble function descriptor

            .text                                 # switch to ".text" (code) section
            .globl   .main                        # make code label visible globally
.main:                                            # code label

# prologue for main
            mflr     r.0                          # get LR into reg 0
            stw      r.0, -4(r.sp)                # save our LR
            stw      r.2, -8(r.sp)                # save our rtoc
            stwu     r.sp, -80(r.1)               # buy stack frame (16 byte boundary)
# line 14
            lwz      r.3, [toc] exint (r.toc)     # load toc entry pointing to exint
            li       r.0, 2                       # load value '2'
            stw      r.0, 0(r.3)                  # store in exint

# line 15
            lwz      r.3, [toc] inint (r.toc)     # load toc entry pointing to inint
            li       r.0, 3                       # load value '3'
            stw      r.0, 0(r.3)                  # store in inint

# line 16
            lwz      r.3, [toc] stint (r.toc)     # load toc entry pointing to stint
            li       r.0, 4                       # load value '4'
            stw      r.0, 0(r.3)                  # store in stint

# line 18
            lwz      r.4, [toc] exproc (r.toc)    # load toc entry pointing to exproc's
                                                  #     function descriptor
            lwz      r.0, 0(r.4)                  # load address of code from fndesc
            mtlr     r.0                          # move code (entry point) addr to LR
            li       r.3, 5                       # load value '5' as 1st parameter
            lwz      r.toc, 4(r.4)                # load toc address for called program
            blrl                                  # branch and link through the LR

            st       r.3, 56(r.sp)                # store rc

# line 20
            li       r.3, 6                       # load value '6' as 1st parameter
            bl       .stproc                      # "local" call to function stproc

# epilogue for main

            lwz      r.0, 68(r.sp)                # reload our saved LR
            mtlr     r.0                          # put return address in LR
            lwz      r.toc, 64(r.sp)              # restore our toc register
            lwz      r.sp, 0(r.sp)                # back to caller's stack frame
```

```
# line 22
              blr                                   # return

# line 25
              .reldata                              # switch to "relocatable data" section
              .globl    inproc                      # make label on fndesc visible globally
inproc:                                             # fndesc name = C function name
              .long     .inproc, .toc, 0            # assemble function descriptor

              .text                                 # switch to ".text" (code) section
.inproc:                                            # trivial program needs no stack frame
              addi      r.3, r.3, 1                 # add 1 to incoming parameter,
              blr                                   #     and return it

# line 26
 .stproc:                                           # trivial program needs no stack frame
              addi      r.3, r.3, 2                 # add 2 to incoming parameter
              blr                                   #     and return it
```

There are some things to note about the "generated code" in this example:

- An individual TOC entry is shown for the static internal variable stint. In fact, all such static data for a given compilation (or for a function, in the case of static data within a function) can be collected together and a TOC entry assigned for the first item only. Once the address of the first item is loaded from the TOC, access to the others can be via fixed displacement from the base pointer.
- Since both the link register and rtoc are modified in main(), locations in the stack frame are established to permit prologue code to save them. The stack frame size is actually 12 bytes more than is required since the frame size must be a multiple of 16 bytes.
- Function main() does not need to restore its TOC pointer immediately after the call to exproc() because it is not required either in main() itself or in the call to stproc(). The linkage conventions require that it be restored prior to the return, however, and a scheduling advantage is gained by restoring it as shown.
- To keep the example as simple as possible, not every possible scheduling optimization has been done to overlap storage accesses with computation.
- A function descriptor for function inproc() is constructed, because the function is visible to separately-compiled modules. No such function descriptor is constructed for function stproc() because that function is static (local to the current compilation) and can be called directly. Doing it this way is a compiler optimization: it would be correct to generate a function descriptor for stproc() that does not have a globally-visible name, and to use it for the call at source line 20. The example, however, is coded to show how a local call, not via function descriptor, works.
- To keep the example simple, the instructions needed to generate the tag table and the tag table index entry have not been shown.

8 APPENDIXES

8.1 Inlining Glue

By default, most compilers assume that local linkage is to be generated for all calls not to function pointers. In AIX compilers, this default can be overridden by compiler command-line arguments and by #pragma statements. The following is a brief description that may be of use to compiler writers.

8.1.1 Compiler command-line arguments

To specify that the default should be to assume local linkage for all calls (this is redundant, as it is already the default), use

```
-qproclocal
```

on the command line when invoking the compiler. To specify that local linkage is to be assumed for calls to specific procedures, use

-qproclocal=*procedure1:procedure2:. . .*

on the command line. That is, specify a list of procedure names separated by colons.

To specify that the default should be to assume linkage through function descriptors for all calls , use

```
-qprocimported
```

on the command line. To specify that linkage through function descriptors is to be assumed for calls to specific procedures, use

-qprocimported=*procedure1:procedure2:. . .*

on the command line.

Both -qproclocal and -qprocimported can be used at once, one with a list of procedures. The scope of both is the entire compilation unit.

8.1.2 Pragmas in source files

Two #pragma statements are provided that produce the same effects as the command-line arguments described above, but that can have a scope limited to just a portion of the compilation. The effect of each of these statements extends from the point at which the statement appears to the end of the source file, unless overridden by a subsequent statement.

To specify that the default should be to assume local linkage for calls to all subsequently declared or defined procedures, use

```
#pragma proclocal
```

To specify that local linkage is to be assumed for calls to specific procedures only, use

#pragma proclocal *procedure1 procedure2 . . .*

That is, specify a list of procedure names separated by white space. This pragma must occur before the declaration or definition of any of the names procedures.

To specify that the default should be to assume linkage through function descriptors for calls to all subsequently declared or defined procedures, use

#pragma procimported

To specify that linkage through function descriptors is to be assumed for calls to specific procedures only, use

#pragma procimported *procedure1 procedure2 . . .*

Again, the named procedure declarations or definitions must occur following this pragma.

Since declarations (function prototypes) and definitions may appear in the same file but not necessarily in the scope of the same pragma, it is the programmer's responsibility to insure that the attributes are consistent.

8.2 LE differences from Power ABI

The Power ABI describes the linkage conventions adopted for big endian PowerPC implementations. These are derived fairly closely from the AIX linkage conventions. This appendix is a list of significant differences between those conventions and the conventions described in this document. Some of the differences result from machine characteristics, and others are felt to result in significant performance improvement.

8.2.1 Alignment

In general, alignment is much more important on early implementations of PowerPC. Therefore, the default mapping rules are somewhat more rigid.

- For NT, all parameters longer than 7 bytes passed in registers must begin on an odd numbered register; and if passed in storage, must be aligned on a double word boundary.

8.2.2 Function descriptors

In AIX and Power, these are 3 words long. In this convention, they are 2 words long. The second word is the anchor for static data addressing. For most procedures, this is the TOC address. For statically nested blocks (like termination handlers) it is the base of addressing for the statically containing block's stack frame. The TOC address is saved in that stack frame so that up-level addressing via this "static back chain" provides addressability to all of static storage.

8.2.3 General register 13 reserved

General register 13 is reserved for system use and must be avoided by compilers.

8.2.4 Different glue sequences

Except for the NT implementation, the glue sequences are essentially the same as they are for AIX. For imported functions, the use of two-instruction glue sequences branching to a common pointer glue sequence (as described in Section 3.6.4) is encouraged to save space. The additional branch is almost always “free” as its target is resolved and the heavy use of the common pointer glue will minimize any cache miss effects.

8.2.5 Modified stack frame layout and parameter passing

The stack frame layout differs from the AIX version in two respects:

- The stack frame header is only 4 words (16 bytes)
- The argument list that is constructed in the caller’s stack frame consists only of the overflow area.

Both modifications help to reduce storage utilization: the minimum stack frame size is only 16 bytes as opposed to 56 in AIX (that would be 64 in PowerPC because of the 16-byte stack alignment requirement).

The larger stack frame sizes in AIX are there to provide space to materialize full parameter lists in storage for functions that take the address of their parameters. These conventions achieve the same function without always reserving the storage in advance, and with a slight increase in code complexity occuring only in functions that use this facility. Thus, the implementaion cost for addressing parameters is borne only by those functions that do so.