IA-64 Runtime Architecture Supplement

Software Conventions for HP-UX

Version 2.6
April 5, 2000

This paper describes software conventions for HP-UX that are not covered by the common runtime conventions. In many cases, the HP-UX compiler tool chain may take advantage of the conventions specified here to produce better object code than what could be achieved through strict conformance to the common conventions.

Introduction

The common runtime document prescribes certain code sequences for accessing data and making procedure calls. In order to support dynamic linkage and maximum addressability, these code sequences are usually not as efficient as they could be if the compiler had additional information about the program as a whole, through extra compiler options or pragmas, or through feedback loops in the compile path. This paper describes a number of improved code sequences that can be generated in the presence of additional information.

Section 1 discusses HP-UX runtime conventions that are not covered in the common runtime document. In many cases, these additional conventions allow compilers to generate better code sequences when compiling object code for HP-UX.

Section 2 presents examples of the code sequences described in this paper. Other example code sequences referred to in this paper can be found in Chapter 9 of the common runtime document.

1. Symbol Binding Model

Symbolic references are bound to symbol definitions at any of three points in the compilation process: at compile time, at link time, and at load time. In general, a binding should be made at the earliest possible point where it is known that the binding is correct. The symbol binding model described here determines when a binding can be made at compile time or link time, and when it must be deferred to load time.
Symbol export class

The ELF object file format contains a symbol export class field that controls the visibility of an external symbol across load modules. The following export classes are defined:

- Default. A local symbol with default export class is not visible outside the load module, and its definition may not be pre-empted; a global (or weak) symbol with default export class is visible and may be pre-empted.

- Hidden. A hidden symbol is not visible outside the load module, regardless of its binding mode (scope), and may not be pre-empted.

- Protected. A protected symbol is visible outside the load module, but may not be pre-empted by a definition in another load module. Any reference to a protected symbol must be resolved to a definition within the same load module.

Protected and hidden symbols are guaranteed to resolve within a load module, and may be treated as “own” symbols.

Both unsatisfied and defined symbols may specify an export class. When linking object files, a resolved symbol will be marked “hidden” if the definition or any reference is marked “hidden”; otherwise, it will be marked “protected” if the definition or any reference is marked “protected.”

If a protected symbol defined in load module A is pre-empted by another definition in load module B, references from within load module A will resolve to the definition in load module A, but references from any other load module where the symbol is not marked “protected” will resolve to the pre-empting definition in load module B.

Binding at compile time

The compiler may bind a reference to a definition at compile time:

- If the definition is contained in the same translation unit as the reference, and is not a tentative definition.

- If the symbol is declared “protected” or “hidden.”

- If the reference is to an inlineable routine when compiling at an appropriate level of optimization (e.g., cross-module inlining).

- If the reference is to a function in the system’s reserved namespace (as determined by the language standard and header files included).

The first rule gives the compiler flexibility to generate better code for references within a translation unit (TU). In a dynamically-bound program, it is possible that any definition within the TU may be pre-empted by another definition in another load module, but the program’s behavior is undefined if, during execution, control flow reaches an entry point in any TU that contains a pre-empted definition. The compiler, linker, and loader are not required to detect this undefined situation, and a reference to the pre-empted symbol from within the same TU may bind either to the visible definition at load time, or to the pre-empted definition.

The second rule allows the compiler to optimize references to symbols that are guaranteed to resolve within the same load module.
The third rule allows the compiler to assume that the inline source visible at compile time is the definition to which a non-inlined call would have bound.

The fourth rule allows the programmer to declare that certain standard C library routines have not been overridden.

**Binding at link time**

The linker may bind a reference to a definition at link time:

- If the linker is building an a.out, and the definition is contained in one of the link units (LUs) submitted as input to the link.
- If the definition is contained in one of the LUs submitted as input to the linker, and is marked as “protected” or “hidden”.

**Binding at load time**

When a program is loaded, the dynamic loader binds all unbound references to definitions based on the binding order of the load modules that are loaded with the program. The binding order is determined by the link order of the DLLs when the program is built, and on the topological order of the dependencies between DLLs. The a.out itself is always the first load module in the binding order, followed by DLLs named on the command line when the program was linked. For each library in the binding order, each of its dependent libraries is added to the binding order if it is not already there. The exact order is not specified here, but a library always precedes its dependent libraries in the binding order. If there is a circular reference, or if the link order conflicts with recorded dependencies, the relative order of the affected libraries is unspecified.

For each unbound reference with default export class, the dynamic loader finds the earliest load module in the binding order that contains a definition to satisfy that reference. (No load module may export two definitions of the same symbol.) Symbols marked “protected” or “hidden” must resolve to a definition in the same load module.

**Compiler options to control binding time**

The following compiler options may be used to control the binding time (some of these options accept a list of symbols or a file name containing a list of symbols):

- `-Bprotected` marks symbols as “protected.” With no list of symbols, this option applies to all symbols defined or referenced in the TU; with a list, it applies only to symbols named in the list.
- `-Bprotected_def` marks symbols defined in the translation unit as “protected.” With no list, this option applies to all defined symbols in the TU; with a list, it applies only to symbols named in the list and defined in the TU. This option does not apply to tentative definitions.
- `-Bhidden` declares the listed symbols as “hidden.” If applied to an unresolved symbol, the attribute will propagate to the definition when the symbol is resolved at link time; if the symbol cannot be bound within the load module, the linker will issue a diagnostic.
• **–Bdefault** applies the default binding attributes to the listed symbols. By default, calls to functions external to the TU use a pc-relative direct call instruction, and save and restore gp around the call; references to data symbols defined external to the TU use the linkage table.

• **–Bextern** asserts that the listed symbols are defined external to the load module. When generating a call to a listed symbol, the compiler should generate an inlined import stub. (See Example 2, below.)

• **–exec** asserts that the code is being compiled for the main program in a dynamically-bound program. This allows the compiler to treat all symbols defined within the TU as “protected.”

• **–noshared** asserts that the code is being compiled for a statically-bound program, which allows the compiler to treat all symbols as “protected,” and allows the use of absolute addressing for text and data. Object files compiled with this option will be marked with the EF_IA_64_CONS_GP flag in the ELF file header. Object files with this flag can be linked with other object files, but the output file must be a statically-bound program file. When object files are linked with the **–r** linker option, the resulting object file must have the EF_IA_64_CONS_GP flag set if the flag is set in any input file.

**Effects of early binding**

When the compiler is allowed to bind a symbol, it may generate code that takes advantage of its knowledge about the definition.

For function calls, the compiler may generate a pc-relative call, omitting the save and restore of the gp register surrounding the call. The call instruction must be tagged with a special relocation, PCREL21BI, that marks the call as “Internal” so the linker will honor the compile-time binding. (See Example 1, below.)

For data declared **const**, the compiler may choose one of the following:

• It may materialize the constant through immediate operands.

• If the constant is no larger than the short data threshold (default 8 bytes), it may place the constant in the short data area, and use a gp-relative addressing sequence to load the value. (See “Addressing own short data” in Chapter 9 of the common conventions document.)

• It may place the constant in the read-only data section and use a pc-relative addressing sequence to load the value. (See “Addressing literals in the text segment” in Chapter 9 of the common conventions document.)

• It may place the constant in the long data area, and use a long-form gp-relative sequence to load the value. (See Example 3, below.)

For an unsatisfied reference to a const data item (which may resolve to either the text or data segment), the compiler must use a linkage table entry unless one of the following conditions is true:

• If the **–noshared** option is in effect, the code may use absolute addressing.

• If the **-exec** option is in effect, and the symbol is declared “protected” or “hidden”, the code may use absolute addressing.
Any use of absolute addressing requires that the EF_IA_64_ABSOLUTE flag be set in the ELF header.

For global variables, the compiler may choose one of the following:

- If the variable is placed in the short data area, it may use a short-form gp-relative addressing sequence to access the variable. (See “Addressing own short data” in Chapter 9 of the common conventions document.)
- If the variable is placed in the long data area, it may use a long-form gp-relative addressing sequence to access the variable. (See Example 3, below.)

The compiler also has the following options when it knows that certain symbols will be bound at link time:

- It may omit the save and restore of the gp register, as described above, if it knows that a call will be bound at link time. As above, the compiler must tag the call with the PCREL21BI relocation. (See Example 1, below.)
- For any reference to “protected” or “hidden” data, the compiler can sometimes infer whether the data is in the short or long data area, based on the type given in the external declaration. If the type is an array type with unspecified size or a Fortran common block, the compiler may not infer the size, and must assume that the data is in the long data area. For other types, the compiler may trust the size of the declared type in determining where the data will be located. If a short data item is declared as “protected” or “hidden” in a TU where referenced, but not in the TU where defined, it may not lie within reach of a short gp-relative addressing sequence. If this situation occurs, the link will fail with an out-of-range relocation error.
- For any reference to data known to be in the text or data segment of the a.out, the compiler may use an absolute addressing sequence to load the value. When it does this, it must tag the object file with the EF_IA_64_ABSOLUTE flag in the ELF header, to ensure that the resulting a.out will not be relocated at load time. (See Example 4, below.)

When the linker processes a pc-relative (except as noted below) or a gp-relative relocation, it binds the reference with a statically-determined pc-relative or gp-relative offset, respectively, so that load-time pre-emption cannot occur.

When the linker is allowed to bind a function call that was compiled with a direct pc-relative call sequence, or if the call was tagged with the PCREL21BI relocation, it will bind the call instruction directly to the target with a statically-determined pc-relative displacement. Otherwise, it must create an import stub and bind the call instruction to the pc-relative displacement of the stub.

When the linker is allowed to bind a data reference that was compiled with LTOFF22X and LDXMOV relocations, it may rewrite the indirect addressing sequence with a direct addressing sequence, as shown in Example 5, below. The rewritten code sequence is the same length as the original, but it replaces a load instruction with a move instruction.

The compiler may use the LTOFF22X and LDXMOV relocations for any reference to long data or non-own data. (It should not use these relocations for short data
that is defined in the same TU, or that is declared “protected” or “hidden,” or
when the \texttt{--exec} option is in effect, since it is better to use a short \texttt{gp-relative}
addressing sequence.) The linker may rewrite the code sequence as long as the
symbol is within the \texttt{gp-relative} addressing range of the \texttt{addl} instruction and
one of the following conditions is true:

- The symbol is marked “protected” or “hidden.”
- The linker is building an executable file (ELF type \texttt{ET\_EXEC}).
- The symbol is defined in the same LU as the reference.

2. Memory Model

In the standard memory model described by the common conventions
document, the maximum size of the short data area is 4 MB, and all linkage
table entries and short own data must fit within this region. In order to
minimize the likelihood of overflowing this area, all data larger than 8 bytes
must be placed in the long data area, even if it is local to the load module.

HP-UX provides three variants of the standard memory model:

- A partitioned model with a user-specified size threshold. The \texttt{+Oshortdata=n}
  compiler option selects a different threshold (minimum 8 bytes) for the
  partitioning of short and long data. All data larger than the specified
  threshold will be placed in long data. To ensure consistent behavior, it is
  advisable to use the same threshold for all translation units in the same
  load module.

- A “tiny” model, where all data is required to fit within the 4 MB limit. In
  this model, equivalent to setting the threshold to 4 MB, all own data may
  be accessed directly with a \texttt{gp-relative} offset, regardless of its size. This
  model can be selected with the \texttt{+Oshortdata} compiler option (with no
  threshold specified). As long as the total size of the short and long data
  areas does not exceed 4 MB, translation units compiled with this option
  may be freely mixed with translation units compiled under the other
  HP-UX memory models.

- A “huge” model, in which the linkage table itself may be larger than 4 MB.
  In this model, the compiler must access all linkage table entries using a
  move long immediate instruction to form the \texttt{gp-relative} offset (see
  Example 6, below); own data must also be accessed with a long-form
direct \texttt{gp-relative} offset, regardless of size (see Example 3, below). No
  linkage table entries are required for long own data. This model is selected
  with the \texttt{+Oshortdata=0} compiler option. To ensure consistent behavior, it is
  advisable to use this model for all translation units in the same load
  module.

3. Data Representation

HP-UX uses the LP64 data model for 64-bit applications (for details on 32-bit
applications, see the separate paper “32-Bit Runtime Architecture for HP-UX”).
In this data model, the \texttt{long} and \texttt{long long} types are the same as \texttt{__int64}, and the
\texttt{long double} type is the same as \texttt{__float128}. 

6 Software Conventions for HP-UX
HP-UX supports the __fpreg type, which is a memory representation of the floating-point register format, as defined by the stf.spill instruction. This type has size and alignment of 16 bytes. The parameter passing and return conventions for __fpreg are the same as for __float80, including the treatment of homogeneous floating-point aggregates (HFAs).

4. Register Usage

The general registers used for return values (r8–r11) may be used for passing language-specific information from a caller to a callee; these registers must not be used as scratch registers by stub routines inserted by the linker or any instrumentation. (This convention is intended for the common runtime document; until it is approved, however, it is considered an HP-UX convention.)

5. ELF Object File Identification

Object files compiled for HP-UX must be marked with one of the following values in the e_ident[EI_OSABI] field in the ELF header:

- ELFOSABI_NONE indicates that the object file conforms to the Unix ABI conventions, and does not use any HP-UX extensions.
- ELFOSABI_HPUX indicates that the object file uses one or more HP or HP-UX extensions to the ELF format.

Note A third value, ELFOSABI_STANDALONE, is deprecated. This value has been used to identify object files compiled for standalone use (e.g., kernels, boot utilities, and diagnostics). This value may be set at link time for compatibility with older tools that require it. The ELF header flags used to mark special conventions are sufficient to mark such objects as different from HP-UX application code.

6. Example Code Sequences

In the sample code sequences in this section, registers of the form t1, t2, etc., are temporary registers, and may be assigned to any available scratch register; registers of the form loc0, loc1, etc., are used for intermediate or final results that may be reused by later code sequences. The code sequences show necessary cycle breaks, but no other scheduling considerations have been made. It is assumed that these code sequences will be scheduled with surrounding code to make best use of the processor resources.

The code sequence in Example 1 is similar to the standard direct procedure call, except that the save and restore of gp is omitted. The @intern() operator forces the use of the PCREL21BI relocation.

Example 1. Direct noextern procedure calls

```
br.call rp = @intern(func) // make the call```
Example 2 shows the code sequence required for an inlined import stub. A requirement specific to the Unix ABI and HP-UX ABI is that the gp value must be copied to r14 so that the caller's gp value can be recovered if the call is diverted to the bind-on-reference routine.

**Example 2. Inlined import stub**

```assembly
mov loc0 = gp  // save current gp
addl t1 = @(ptoff(func), gp)  // calc. address of LT entry
;;
l8 t2 = [t1], 8  // load entry point
mov r14 = gp  // copy gp for BOR routine
;;
l8 gp = [t1]  // load new gp value
mov b6 = t2  // move ep to call BR
br.call rp = b6  // make the call
;;
mov gp = loc0  // restore gp
```

Example 3 shows the code sequence that can be used when addressing long own data.

**Example 3. Long-form gp-relative addressing**

```assembly
movl t1 = @gprel(var)  // form gp-relative offset
;;
add t2 = t1, gp  // add it to gp
;;
l8 loc0 = [t2]  // load the variable
```

Example 4 shows the code sequence that can be used to access data using absolute addressing.

**Example 4. Absolute addressing**

```assembly
movl t1 = var  // get address of var
;;
l8 loc0 = [t1]  // load contents of var
```

Example 5 shows the default code sequence recommended by the Unix ABI for addressing external variables through the linkage table. This sequence uses the special relocations designed to allow the linker to rewrite the first and second instructions to eliminate the second load. The @ltoffx() operator translates to the LTOFF22X relocation, and the ldxmov pseudo-op translates to an ld8 instruction with the LDXMOV relocation. If the variable proves to be external, the linker processes the LTOFF22X relocation as LTOFF22, and ignores the LDXMOV relocation. If the variable is bound within the load module, the linker processes the LTOFF22X relocation as GPREL22, and processes the LDXMOV relocation by translating the ld8 t2 = [t1] to mov t2 = t1.
Note that two code paths referencing different variables may not join at a common ldxmov, since the decision to convert the load to a move is based on the location of the variable being referenced. Either the ldxmov must be placed on each path, or the use of these special relocations must be avoided in such a case.

**Example 5. Rewritable indirect addressing**

```asm
addl t1 = @ltoffx(var), gp // calc. address of LT entry
;;
ldxmov t2 = [t1], var // load address of var
;;
ld8 loc0 = [t2] // load contents of var
```

Example 6 shows the code sequence to be used to address external data in the huge model.

**Example 6. Indirect addressing of data in the huge model**

```asm
movl t1 = @ltoff(var) // form gp-rel offset to LT entry
;;
add t2 = t1, gp // add it to gp
;;
ld8 t3 = [t2] // load the LT entry
;;
ld8 loc0 = [t3] // load contents of var
```