NAME

libunwind-dynamic -- libunwind-support for runtime-generated code

INTRODUCTION

For libunwind to do its job, it needs to be able to reconstruct the *frame state* of each frame in a call-chain. The frame state describes the subset of the machine-state that consists of the *frame registers* (typically the instruction-pointer and the stack-pointer) and all callee-saved registers (preserved registers). The frame state describes each register either by providing its current value (for frame registers) or by providing the location at which the current value is stored (callee-saved registers).

For statically generated code, the compiler normally takes care of emitting *unwind-info* which provides the minimum amount of information needed to reconstruct the frame-state for each instruction in a procedure. For dynamically generated code, the runtime code generator must use the dynamic unwind-info interface provided by libunwind to supply the equivalent information. This manual page describes the format of this information in detail.

For the purpose of this discussion, a *procedure* is defined to be an arbitrary piece of *contiguous* code. Normally, each procedure directly corresponds to a function in the source-language but this is not strictly required. For example, a runtime code-generator could translate a given function into two separate (discontiguous) procedures: one for frequently-executed (hot) code and one for rarely-executed (cold) code. Similarly, simple source-language functions (usually leaf functions) may get translated into code for which the default unwind-conventions apply and for such code, it is not strictly necessary to register dynamic unwind-info.

A procedure logically consists of a sequence of *regions*. Regions are nested in the sense that the frame state at the end of one region is, by default, assumed to be the frame state for the next region. Each region is thought of as being divided into a *prologue*, a *body*, and an *epilogue*. Each of them can be empty. If non-empty, the prologue sets up the frame state for the body. For example, the prologue may need to allocate some space on the stack and save certain callee-saved registers. The body performs the actual work of the procedure but does not change the frame state in any way. If non-empty, the epilogue restores the previous frame state and as such it undoes or cancels the effect of the prologue. In fact, a single epilogue may undo the effect of the prologues of several (nested) regions.

We should point out that even though the prologue, body, and epilogue are logically separate entities, optimizing code-generators will generally interleave instructions from all three entities. For this reason, the dynamic unwind-info interface of libunwind makes no distinction whatsoever between prologue and body. Similarly, the exact set of instructions that make up an epilogue is also irrelevant. The only point in the epilogue that needs to be described explicitly by the dynamic unwind-info is the point at which the stack-pointer gets restored. The reason this point needs to be described is that once the stack-pointer is restored, all values saved in the deallocated portion of the stack frame become invalid

and hence libunwind needs to know about it. The portion of the frame state not saved on the stack is assumed to remain valid through the end of the region. For this reason, there is usually no need to describe instructions which restore the contents of callee-saved registers.

Within a region, each instruction that affects the frame state in some fashion needs to be described with an operation descriptor. For this purpose, each instruction in the region is assigned a unique index. Exactly how this index is derived depends on the architecture. For example, on RISC and EPIC-style architecture, instructions have a fixed size so it's possible to simply number the instructions. In contrast, most CISC use variable-length instruction encodings, so it is usually necessary to use a byte-offset as the index. Given the instruction index, the operation descriptor specifies the effect of the instruction in an abstract manner. For example, it might express that the instruction stores callee-saved register r1 at offset 16 in the stack frame.

PROCEDURES

A runtime code-generator registers the dynamic unwind-info of a procedure by setting up a structure of type unw_dyn_info_t and calling _U_dyn_register(), passing the address of the structure as the sole argument. The members of the unw_dyn_info_t structure are described below:

void *next
: Private to libunwind. Must not be used by the application.

void *prev
: Private to libunwind. Must not be used by the application.

unw_word_t start_ip
: The start-address of the instructions of the procedure (remember: procedure are defined to be contiguous pieces of code, so a single code-range is sufficient).

unw_word_t end_ip
: The end-address of the instructions of the procedure (non-inclusive, that is, end_ip-start_ip is the size of the procedure in bytes).

unw_word_t gp
: The global-pointer value in use for this procedure. The exact meaning of the global-pointer is architecture-specific and on some architecture, it is not used at all.

int32_t format
: The format of the unwind-info. This member can be one of UNW_INFO_FORMAT_DYNAMIC, UNW_INFO_FORMAT_TABLE, or UNW_INFO_FORMAT_REMOTE_TABLE.

union u
This union contains one sub-member structure for every possible unwind-info format:

unw_dyn_proc_info_t pi
This member is used for format UNW_INFO_FORMAT_DYNAMIC.

unw_dyn_table_info_t ti
This member is used for format UNW_INFO_FORMAT_TABLE.

unw_dyn_remote_table_info_t rti
This member is used for format UNW_INFO_FORMAT_REMOTE_TABLE.

The format of these sub-members is described in detail below.

PROC-INFO FORMAT

This is the preferred dynamic unwind-info format and it is generally the one used by full-blown runtime code generators. In this format, the details of a procedure are described by a structure of type unw_dyn_proc_info_t. This structure contains the following members:

unw_word_t name_ptr
The address of a (human-readable) name of the procedure or 0 if no such name is available. If non-zero, the string stored at this address must be ASCII NUL terminated. For source languages that use name mangling (such as C++ or Java) the string stored at this address should be the *demangled* version of the name.

unw_word_t handler
The address of the personality routine for this procedure. Personality routines are used in conjunction with exception handling. See the C++ ABI draft (http://www.codesourcery.com/cxx-abi/) for an overview and a description of the personality routine. If the procedure has no personality routine, handler must be set to 0.

uint32_t flags
A bitmask of flags. At the moment, no flags have been defined and this member must be set to 0.

unw_dyn_region_info_t *regions
A NULL-terminated linked list of region descriptors. See section ‘‘Region descriptors’’ below for more details.

TABLE-INFO FORMAT

This format is generally used when the dynamically generated code was derived from static code and

the unwind-info for the dynamic and the static versions are identical. For example, this format can be useful when loading statically-generated code into an address-space in a non-standard fashion (i.e., through some means other than dlopen()). In this format, the details of a group of procedures is described by a structure of type unw_dyn_table_info. This structure contains the following members:

unw_word_t name_ptr
The address of a (human-readable) name of the procedure or 0 if no such name is available. If non-zero, the string stored at this address must be ASCII NUL terminated. For source languages that use name-mangling (such as C++ or Java) the string stored at this address should be the *demangled* version of the name.

unw_word_t segbase
The segment-base value that needs to be added to the segment-relative values stored in the unwind-info. The exact meaning of this value is architecture-specific.

unw_word_t table_len
The length of the unwind-info (table_data) counted in units of words (unw_word_t).

unw_word_t table_data
A pointer to the actual data encoding the unwind info. The exact format is architecture-specific (see architecture-specific sections below).

REMOTE TABLE-INFO FORMAT

The remote table-info format has the same basic purpose as the regular table-info format. The only difference is that when libunwind uses the unwind-info, it will keep the table data in the target address-space (which may be remote). Consequently, the type of the table_data member is unw_word_t rather than a pointer. This implies that libunwind will have to access the table-data via the address-space's access_mem() call-back, rather than through a direct memory reference.

From the point of view of a runtime code generator, the remote table-info format offers no advantage and it is expected that such generators will describe their procedures either with the proc-info format or the normal table-info format. The main reason that the remote table-info format exists is to enable the address-space-specific find_proc_info() callback (see unw_create_addr_space(3libunwind)) to return unwind tables whose data remains in remote memory. This can speed up unwinding (e.g., for a debugger) because it reduces the amount of data that needs to be loaded from remote memory.

REGIONS DESCRIPTORS

A region descriptor is a variable length structure that describes how each instruction in the region affects the frame state. Of course, most instructions in a region usually do not change the frame state and for those, nothing needs to be recorded in the region descriptor. A region descriptor is a structure

of type unw_dyn_region_info_t and has the following members:

unw_dyn_region_info_t *next
A pointer to the next region. If this is the last region, next is NULL.

int32_t insn_count
The length of the region in instructions. Each instruction is assumed to have a fixed size (see architecture-specific sections for details). The value of insn_count may be negative in the last region of a procedure (i.e., it may be negative only if next is NULL). A negative value indicates that the region covers the last *N* instructions of the procedure, where *N* is the absolute value of insn_count.

uint32_t op_count
The (allocated) length of the op_count array.

unw_dyn_op_t op
An array of dynamic unwind directives. See Section ‘‘Dynamic unwind directives’’ for a description of the directives.

A region descriptor with an insn_count of zero is an *empty region* and such regions are perfectly legal. In fact, empty regions can be useful to establish a particular frame state before the start of another region.

A single region list can be shared across multiple procedures provided those procedures share a common prologue and epilogue (their bodies may differ, of course). Normally, such procedures consist of a canned prologue, the body, and a canned epilogue. This could be described by two regions: one covering the prologue and one covering the epilogue. Since the body length is variable, the latter region would need to specify a negative value in insn_count such that libunwind knows that the region covers the end of the procedure (up to the address specified by end_ip).

The region descriptor is a variable length structure to make it possible to allocate all the necessary memory with a single memory-allocation request. To facilitate the allocation of a region descriptors libunwind provides a helper routine with the following synopsis:

size_t _U_dyn_region_size(int op_count);

This routine returns the number of bytes needed to hold a region descriptor with space for op_count unwind directives. Note that the length of the op array does not have to match exactly with the number of directives in a region. Instead, it is sufficient if the op array contains at least as many entries as there are directives, since the end of the directives can always be indicated with the UNW_DYN_STOP

directive.

DYNAMIC UNWIND DIRECTIVES

A dynamic unwind directive describes how the frame state changes at a particular point within a region. The description is in the form of a structure of type unw_dyn_op_t. This structure has the following members:

int8_t tag
: The operation tag. Must be one of the unw_dyn_operation_t values described below.

int8_t qp
: The qualifying predicate that controls whether or not this directive is active. This is useful for predicated architectures such as IA-64 or ARM, where the contents of another (callee-saved) register determines whether or not an instruction is executed (takes effect). If the directive is always active, this member should be set to the manifest constant _U_QP_TRUE (this constant is defined for all architectures, predicated or not).

int16_t reg
: The number of the register affected by the instruction.

int32_t when
: The region-relative number of the instruction to which this directive applies. For example, a value of 0 means that the effect described by this directive has taken place once the first instruction in the region has executed.

unw_word_t val
: The value to be applied by the operation tag. The exact meaning of this value varies by tag. See Section ‘‘Operation tags’’ below.

It is perfectly legitimate to specify multiple dynamic unwind directives with the same when value, if a particular instruction has a complex effect on the frame state.

Empty regions by definition contain no actual instructions and as such the directives are not tied to a particular instruction. By convention, the when member should be set to 0, however.

There is no need for the dynamic unwind directives to appear in order of increasing when values. If the directives happen to be sorted in that order, it may result in slightly faster execution, but a runtime code-generator should not go to extra lengths just to ensure that the directives are sorted.

IMPLEMENTATION NOTE: should libunwind implementations for certain architectures prefer the

list of unwind directives to be sorted, it is recommended that such implementations first check whether the list happens to be sorted already and, if not, sort the directives explicitly before the first use. With this approach, the overhead of explicit sorting is only paid when there is a real benefit and if the runtime code-generator happens to generate sorted lists naturally, the performance penalty is limited to a simple O(N) check.

OPERATIONS TAGS

The possible operation tags are defined by enumeration type unw_dyn_operation_t which defines the following values:

UNW_DYN_STOP

Marks the end of the dynamic unwind directive list. All remaining entries in the op array of the region-descriptor are ignored. This tag is guaranteed to have a value of 0.

UNW_DYN_SAVE_REG

Marks an instruction which saves register reg to register val.

UNW_DYN_SPILL_FP_REL

Marks an instruction which spills register reg to a frame-pointer-relative location. The frame-pointer-relative offset is given by the value stored in member val. See the architecture-specific sections for a description of the stack frame layout.

UNW_DYN_SPILL_SP_REL

Marks an instruction which spills register reg to a stack-pointer-relative location. The stack-pointer-relative offset is given by the value stored in member val. See the architecture-specific sections for a description of the stack frame layout.

UNW_DYN_ADD

Marks an instruction which adds the constant value val to register reg. To add subtract a constant value, store the two’s-complement of the value in val. The set of registers that can be specified for this tag is described in the architecture-specific sections below.

UNW_DYN_POP_FRAMES

.PP

UNW_DYN_LABEL_STATE

.PP

UNW_DYN_COPY_STATE

.PP

UNW_DYN_ALIAS
.PP unw_dyn_op_t

_U_dyn_op_save_reg(); _U_dyn_op_spill_fp_rel(); _U_dyn_op_spill_sp_rel(); _U_dyn_op_add(); _U_dyn_op_pop_frames(); _U_dyn_op_label_state(); _U_dyn_op_copy_state(); _U_dyn_op_alias(); _U_dyn_op_stop();

IA-64 SPECIFICS

- meaning of segbase member in table-info/table-remote-info format - format of table_data in table-info/table-remote-info format - instruction size: each bundle is counted as 3 instructions, regardless of template (MLX) - describe stack-frame layout, especially with regards to sp-relative and fp-relative addressing - UNW_DYN_ADD can only add to ‘‘sp’’ (always a negative value); use POP_FRAMES otherwise

SEE ALSO

libunwind(3libunwind), _U_dyn_register(3libunwind), _U_dyn_cancel(3libunwind)

AUTHOR

David Mosberger-Tang
Email: **dmosberger@gmail.com**
WWW: **http://www.nongnu.org/libunwind/**.