Compact Exception Tables for MIPS ABIs

Owner: Nathan Sidwell

Changes:

24th June 2011	Initial version
30th June 2011	Clarifications, GNU_EH_FRAME segment compatibility
25th September 2012	Clarifications and adjustments during implementation.
22nd April 2013	Final changes.
15th May 2013	Clarify and fix confusion between unwind-continue and no-throw.
24th November 2015	Definition of additional opcodes. Mention that versioning information may be defined in the header.

1 Introduction and Scope

This document specifies an encoding of exception unwind information necessary to support C++. The desired outcome is an encoding that is denser than the generic DWARF-based scheme currently implemented by GCC.

Guidance is taken from the ARM EABI exception handling ABI. While this document is specifically targeted at MIPS, the C++ unwind encoding could be readily used for other targets with minimal changes.

1.1 Backwards compatibility

The compact scheme must be backwards compatible with the DWARF scheme at the shared object level. Namely, although a shared object must contain either compact tables or DWARF tables, (but not both), a process may consist of shared objects and main executable using a mixture of formats.

As will be seen in Section 9, intermixing within a single translation unit is achieved.

1.2 ABI & ISA compatibility

The compact scheme must be applicable to all current MIPS ABIs and ISAs. Namely:

- o32, n32, o64 & n64 ABIs
- PIC and non-PIC executables
- MIPS16, MIPS32, MIPS64 and microMIPS ISAs

Both bare-metal (static, unhosted) and Linux-hosted (static or dynamically linked) executables and shared objects must be supported.

As noted in Section 2, 64 bit executables and shared objects are limited to 2GB text and data segments (per object).

1.3 Toolchain compatibility

Compilers have considerable freedom in how to layout a stack frame and record callee-save register values. Although the design should consider alternative layouts, the layout as generated by GCC shall be taken as the canonical organization.

1.4 Language support

C++ exception handling is supported, including forced unwinding required for Posix thread cancelation.

1.5 Advanced features

GCC supports two advanced features of unwinding:

- Non-call exceptions
- Asynchronous backtracing

Neither of these shall be considered (the ARM EABI does not support them). They require unwinding information to be correct at every exception raising instruction or at every instruction respectively, and this will significantly bloat the unwinding tables. The former is required for Java support and the latter for providing accurate backtraces (generally upon program termination). Neither is enabled by default. Backtraces can be generated by using libunwind (although that may require modification to read MIPS DWARF debug sections).

If such features are required in a dynamic object, the old DWARF scheme can still be used.

1.6 Desired Toolchain Impact

The compact scheme will require extensive toolchain changes. The compiler, assembler, linker and runtime will require modification. The debugger will not need changing, as it has no inbuilt knowledge of the exception handling scheme, beyond the knowing the name of the entry and exit points.

As documented in Section 11, a new static relocation is defined but no new dynamic relocations are needed.

See Section 12 for a description of implementation complexity within the GNU toolchain.

2 Design Principles and Assumptions

Throwing a C++ exception is expected to be a rare event. The exception handling code should never be the hot-path of an executable. As such, it is always better to design for compact encoding at the expense of speed handling the exception.

Function prologues (and epilogues) are atomic from the POV of unwinding. It is never necessary to unwind a partially constructed stack frame. The 'shrink wrap' optimization, which moves the construction of a stack frame into the body of a function, does not affect this assumption. The outer code is not exception raising, and the prologue remains atomic.

Divergence from the DWARF scheme shall be minimized. This will reduce the number of changes necessary throughout the toolchain. It will also decrease the probability of design errors. In addition, it may allow intermixing of DWARF and compact encoding within a single shared object or even translation unit (although neither is a design requirement).

In common with the DWARF scheme and the ARM EABI design, it shall be possible to mix object files created by different toolchains, but without completely specifying the interface between the runtime and the generated code. This is achieved by using toolchain-specific 'personality' routines. A code generator specifies the personality routine to use when unwinding, and this personality routine has knowledge of the code generation scheme. Personality routines are specified on a per-function basis.

Although not specified in the psABI, static relocations for 32-bit quantities need not be naturally aligned. Dynamic relocations must be aligned.

Dynamic objects have less than 2GB of text segment and 2GB of data segment, even in 64-bit ABIs.

3 ISA and ABI Notes

The MIPS ISAs of interest align instructions to 2 or 4 byte boundaries. ISAs that support two modes of operation use bit 0 to indicate which mode is being executed. However, when referring to code ranges, this bit is free to use for other purposes. Bit 0 only needs setting correctly when jumping to target code.

The MIPS ISAs support both big and little endian data representations.

The more recent ISAs have store multiple instructions designed for creating stack frames. These restrict the freedom a code generator has in selecting which registers to spill. The ABI defines the following convention for integer registers:

Register number	Name	Saved by	Use
$0	$zero		Always zero
$1	at		Assembler temp
$2-$3	v0-v1	caller	Function return
$4-$7	a0-a3	caller	Function args
$8-$15	t0-t7	caller	Temp regs
$16-$23	s0-s7	callee	Saved regs
$24-$25	t8-t9	caller	Temp regs
$26-$27	kt0-kt1		Kernel use
$28	gp	callee	Global pointer
$29	sp		Stack pointer
$30	s8	callee	Saved reg
$31	ra		Return address
	hi, lo	caller	Multiply/divide

GCC allocates callee saves in order $16..$23,$30.

For floating point code, the following register convention is defined:

Register	Saved by	Use
$f0-$f3	caller	Function return
$f4-$f11	caller	Temp regs

Register	Saved by	Use
$f12-$f15	caller	Function args
$f16-$f19	caller	Temp regs
$f20-$f31	callee	Saved regs
fcr31	caller	Control register

Code generators use generic coprocessor load & store instructions to save and restore these registers in prologue and epilogue.

Other registers are caller-saved.

The stack maintains 16 byte or 8 byte alignment for new and old ABIs respectively.

4 References

DWARF 4 specification http://dwarfstd.org/Dwarf4Std.php

MIPS RISC Processor Supplement 3rd Edition SYSTEM V , Application Binary Interface http://www.sco.com/developers/devspecs/mipsabi.pdf

64-bit ELF Object File Specification ftp://ftp.linux-mips.org/pub/linux/mips/doc/ABI/elf64-2.4.pdf

MIPS32 Volume I: Introduction to the MIPS32 Architecture http://www.mips.com/secure-download/index.dot?product_name=/auth/MD00082%2D2B%2DMIPS32INT%2DAFP%2D03.02.pdf

MIPS32 Volume I-B: Introduction to the microMIPS32 Architecture http://www.mips.com/secure-download/index.dot?product_name=/auth/MD00741-2B-microMIPS32INT-AFP-03.02.pdf

MIPS32 Volume II: The MIPS32 Instruction Set http://www.mips.com/secure-download/index.dot?product_name=/auth/MD00086%2D2B%2DMIPS32BIS%2DAFP%2D03.02.pdf

MIPS32 Volume II-B: The microMIPS32 Instruction Set http://www.mips.com/secure-download/index.dot?product_name=/auth/MD00582%2D2B%2DmicroMIPS%2DAFP%2D03.03.pdf

MIPS64 Volume I: Introduction to the MIPS64 Architecture http://www.mips.com/secure-download/index.dot?product_name=/auth/MD00083-2B-MIPS64INT-AFP-05.01.pdf

MIPS64 Volume I-B: Introduction to the microMIPS64 Architecture http://www.mips.com/secure-download/index.dot?product_name=/auth/MD00743-2B-microMIPS64INT-AFP-05.01.pdf

MIPS64 Volume II: The MIPS64 Instruction Set http://www.mips.com/secure-download/index.dot?product_name=/auth/MD00087-2B-MIPS64BIS-AFP-05.01.pdf

MIPS64 Volume II-B: The microMIPS64 Instruction Set

MIPS16e Extension to the MIPS32 Architecture http://www.mips.com/secure-download/index.dot?product_name=/auth/MD00091-2B-MIPS64PRA-AFP-05.01.pdf

MIPS16e Extension to the MIPS64 Architecture http://www.mips.com/secure-download/index.dot?product_name=/auth/MD00077%2D2B%2DMIPS1664%2DAFP%2D02%5F60.pdf

Exception handling ABI for ARM architecture http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.swdev.abi/index.html

C++ ABI https://github.com/MentorEmbedded/cxx-abi

Ian Taylor's descriptions of unwinding http://www.airs.com/blog/archives/460, http://www.airs.com/blog/archives/462 & http://www.airs.com/blog/archives/464 .

5 Abbreviations

CFA Canonical Frame Address. A stable point in the stack frame from whence saved register locations are specified. It is usually stack pointer value at the call site from whence the current function was called.

CIE Common Information Entry. Information common to multiple frames. A CIE is followed by the FDEs to which it is common. There may be multiple CIEs in a translation unit.

FDE Frame Description Entry. Information describing a particular function's stack frame.

LSDA Language Specific Data Area

6 General Description

A C++ exception is generated by a 'throw' expression. This creates an object to be thrown, and unwinds the call stack to locate the most recently entered 'try' statement able to catch the thrown object. During the unwinding process cleanups must be run to destroy objects going out of scope and any function exception specifications need to be checked.

Exceptions may only be generated at call sites (the 'throw' expression ultimately results in a call to the runtime to propagate the exception). Thus the following observations can be made:

- Unwind information only need specify the location of callee-save registers, along with the size of the stack frame.
- Leaf functions are always exception free.

The process of unwinding involves determining the register values at a catch, cleanup or exception check by updating the current register set using unwinding information located in the executable. For each stage of unwinding the following steps occur

1. Construct a scratch context of register values by saving the current register set somewhere.
2. Locate the unwinding table corresponding to the saved PC value. Typically this will be a 2-stage lookup to first determine the shared object containing the saved PC, then searching that shared object's unwind tables for the correct one.
3. Call the personality routine specified for the located unwinding table. The personality routine will return:
    1. Continue unwinding (this is the default, if there is no personality routine, but is an unwind table). It indicates the routine has no cleanups, catches or exception specification to check. The unwinder should consult the unwinding table to determine the location of any callee-save registers, and how to restore the stack pointer and return address. Then continue at step #2 above.
    2. Stop unwinding and call terminate or unexpected (this is the default if there is no unwind table). This indicates a fatal error of unwinding machinery, typically throwing an exception in a region unprepared to deal with it, or violation of an exception specification.
    3. Install context. This indicates a so-called landing pad has been located, and should be executed. The context will be updated to pass zero or more values from the personality routine to the landing pad. The unwinder should install the current context, which (a) releases any previously unwound stack and (b) starts the landing pad code.
4. The landing pad code will either be:
    1. A catch clause that swallows the exception, eventually resulting in a call to finish the

exception handling.

2. A catch clause that rethrows the exception, essentially resulting in the same behaviour as a cleanup
3. A cleanup clause ending with a call to continue unwinding.

5. To continue unwinding return to step #3 above

Note that the vendor neutral ABI actually specifies a two-phase unwinding scheme. The call stack is first tentatively unwound looking for an exception specification or matching catch clause. When one is located, the stack is properly unwound, executing cleanups until the exception specification or catch clause is reached. This allows better diagnostics in the face of failure because the originating throw context is still available. However it means that personality routines have two modes of execution, and are encouraged to cache information about the ultimate landing pad determined in the first phase for use in the second phase.

As can be seen from the description unwind tables comprise 3 separate parts:

1. A mechanism to locate the required unwinding table. Typically a binary search will suffice at the second stage of this lookup.
2. An unwind table describing how to restore the context from one function to its caller. This table is language and toolchain independent.
3. An action table describing the cleanups, catches and exception specification(s) within a particular function. This table is toolchain dependent. Note there may be multiple exception specification regions, introduced by function inlining. Similarly there may be multiple nested catch clauses either written directly by the programmer, or by inlining.

7 DWARF Scheme

The DWARF-based scheme uses unwinding information built from DWARF opcodes. These are not designed for compact encoding.

The unwind table is emitted, via .cfi assembler directives, to a '.eh_frame' section. It consists of a series of FDEs (Frame Descriptor Entries) interspersed with CIEs (Common Information Entries). Each FDE references one CIE, and a CIE may be shared between multiple FDEs. The CIE is a simple compression scheme containing the common prefix of a number of FDEs. The FDEs encode how to restore the context from one stack frame to its parent, this typically means specifying how to determine the CFA (Canonical Frame Address) and the location of all callee-saved registers. The DWARF scheme is expressive enough to accurately describe this restoration at every instruction in a function.

Both FDEs and CIEs are about 24 bytes for MIPS (including the length header). There are a handful of CIEs per object file.

Each FDE that describes a function with cleanups, catches or exception specifications will contain a pointer to a Language Specific Data Area (LSDA).

The LSDA is emitted to a '.gcc_except_table' section. It consists of

- An optional Landing Pad Start address (defaulting to the function start address),
- A few bytes describing the format of the remaining pieces of the LSDA
- A table of address, length, landing pad and action table offsets
- An action table of byte codes

- A table of type info pointers

The main portion describes exception regions within the function (using offsets from the beginning of the function). These exception regions do not overlap. Each entry is a tuple of 4 values:

- Region start offset
- Region length
- Landing pad offset, or zero. Zero indicates a no-exception region.
- Action table offset, or zero. Zero indicates a cleanup.

All four values of each entry are uleb128 encoded and are thus typically one or two bytes each.

The action table is used for exception specifications and catch clauses. It specifies a set of types that may be caught or passed, using the indirection of the type table. An action is one of:

- Index into the type table. This is a catch.
- Offset to zero-terminated list of indexes into the type table. This is an exception specification.
- Zero. This is a clean-up.

Entries in the action table are chained together via a trailing offset value for each action.

The type table typically consists of addresses or offsets to pointers to type_info objects. A value of zero is used for a catch-all handler. This additional layer of indirection is necessary so that the type table can be read-only, but the comdat semantics of type_info objects can be catered for by setting the value of the indirected pointer correctly upon program loading. Weak hidden linkage is taken advantage of to enable only one instance of such an indirect pointer in a shared object. For static executables, such indirection can be omitted, and the data format of the action table will be adjusted.

The unwind table is structured so that tables from different object files, linked together can simply be concatenated, and a zero word appended to the end. At program start up, the unwind table is registered with the runtime via a constructor function. This determines the range of addresses covered by the table. During exception handling, a sorted index table of pointers to FDEs is created, to allow a binary search to discover the FDE of interest. So the dynamic cost is an additional 4 bytes per function.

Both the .eh_frame and the .gcc_except_table sections can be placed in the read-only segment of a dynamic object. Indirection via locations in the data section is used to refer to addresses that must be determined during dynamic loading.

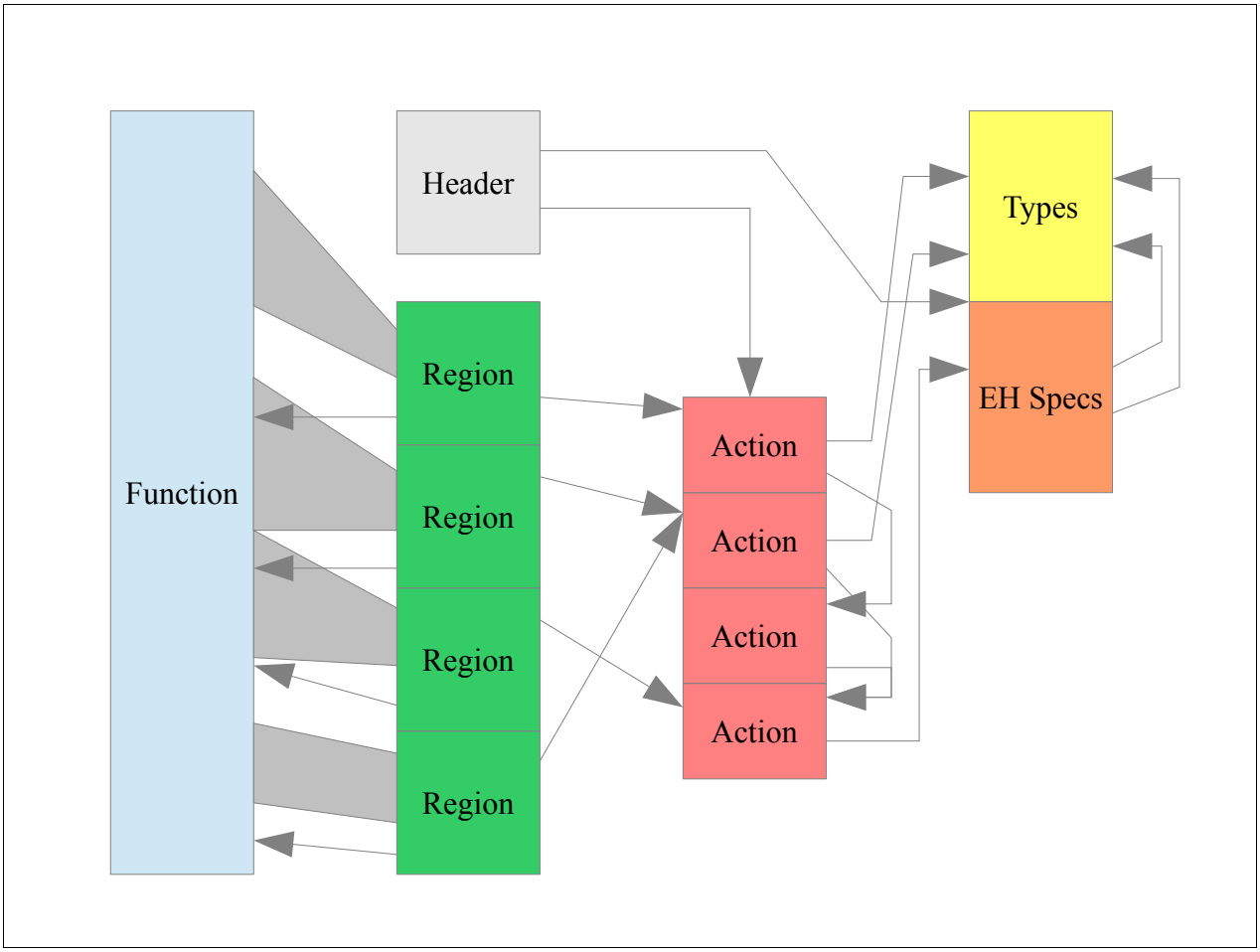
Ian Taylor's blog has a good description of the C++ scheme and DWARF encoding.

7.1.1 Data Layout

The following diagram exemplifies the DWARF scheme:

- All of the pointers are relative offsets. The base of the offset is some other defined point in the data structure, or from the offset itself (i.e. Self-relative).
- NULL entries are an appropriate zero.
- The type table (yellow) and EH spec table (orange) abut.
- The region table (green) and action table (red) abut [not shown in diagram].

- A header (grey) points to the meeting point of the region and action tables, and the meeting point of the type and EH spec tables.
- The EH spec table points to entries in the type table. The end of an EH spec sequence is indicated by a NULL entry.
- Each region table entry indicates a unique code range (grey polygons). These might not abut. Code regions that are not covered are either incapable of raising an exception, or are not permitted to (if they do, it is a user error). The region's start is an offset from the start of the function.
- Each region table entry indicates a landing pad in the code. This offset is relative to the start of the function. This is often within the code range of another region. Sometimes the landing pad is NULL (not shown). This landing pad is responsible for processing any action for exceptions emanating from the region's code range.
- Each region table entry points to an action table entry. Action table entries can be shared (but usually are not).
- Each action table entry points to a successor action table entry. The end of the chain indicates the exception is unhandled and unwinding should continue to the calling context.
- Each action table entry points to either
    - a specific type (to catch that type). A specific type can be shared between multiple action table entries (but usually are not).
    - an EH spec, to check an EH specification. A specific EH spec can be shared, but usually is not.
    - or is NULL to indicate a cleanup.
- If the action matches, the landing pad is called. Information about what matched is passed to the landing pad.
- If the action fails to match, the chain is followed and the next action is checked.



7.2 Coding Example

The following example program fragment contains an inlineable function with an exception specification and a main routine with two sequential nested try...catch clauses.

```
void Foo ();
void C () throw ();

struct X {~X ();};
struct Y {~Y () throw ();};
struct Z {~Z () throw ();};

inline void Baz () throw (int, char *) {
  Foo ();
}

void Bar () throw (int *, void *) {
  { X x;
    try {
      Y y;
```

```
      try {
        Foo ();
      } catch (int i) {
        C ();
      }
      Foo ();
    } catch (float f) {
      Baz ();
    }
  }

  try {
    try {
      Foo ();
    } catch (float f) {
      C ();
    }
    Foo ();
  } catch (int i) {
    Foo ();
  }
}
```

When compiled with optimization (to inline the inlineable function) its DWARF exception information is as follows ($LEHBn, $LEHEn, $LFBn are locations in the emitted code):

```
$LLSDA1:
        .byte   0xff     # @LPStart format (omit)
        .byte   0xbb     # @TType format (indirect datarel sdata4)
        .uleb128 $LLSDATT1-$LLSDATTD1       # @TType base offset
$LLSDATTD1:
        .byte   0x1      # call-site format (uleb128)
        .uleb128 $LLSDACSE1-$LLSDACSB1   # Call-site table length
$LLSDACSB1:
        .uleb128 $LEHB0-$LFB1    # region 0 start
```

```
.uleb128 $LEHE0-$LEHB0          # length
.uleb128 $L26-$LFB1             # landing pad
.uleb128 0x7                    # action
.uleb128 $LEHB1-$LFB1   # region 1 start
.uleb128 $LEHE1-$LEHB1          # length
.uleb128 $L27-$LFB1             # landing pad
.uleb128 0x5                    # action
.uleb128 $LEHB2-$LFB1   # region 2 start
.uleb128 $LEHE2-$LEHB2          # length
.uleb128 $L29-$LFB1             # landing pad
.uleb128 0x1                    # action
.uleb128 $LEHB3-$LFB1   # region 3 start
.uleb128 $LEHE3-$LEHB3          # length
.uleb128 $L30-$LFB1             # landing pad
.uleb128 0xb                    # action
.uleb128 $LEHB4-$LFB1   # region 4 start
.uleb128 $LEHE4-$LEHB4          # length
.uleb128 $L31-$LFB1             # landing pad
.uleb128 0x9                    # action
.uleb128 $LEHB5-$LFB1   # region 5 start
.uleb128 $LEHE5-$LEHB5          # length
.uleb128 0                      # landing pad
.uleb128 0                      # action
.uleb128 $LEHB6-$LFB1   # region 6 start
.uleb128 $LEHE6-$LEHB6          # length
.uleb128 $L32-$LFB1             # landing pad
.uleb128 0x3                    # action
.uleb128 $LEHB7-$LFB1   # region 7 start
.uleb128 $LEHE7-$LEHB7          # length
.uleb128 0                      # landing pad
.uleb128 0                      # action
.uleb128 $LEHB8-$LFB1   # region 8 start
.uleb128 $LEHE8-$LEHB8          # length
```

```
        .uleb128 $L33-$LFB1             # landing pad
        .uleb128 0xd                    # action
        .uleb128 $LEHB9-$LFB1   # region 9 start
        .uleb128 $LEHE9-$LEHB9          # length
        .uleb128 $L28-$LFB1             # landing pad
        .uleb128 0x3                    # action

$LLSDACSE1:         # Action record table
        .byte   0x7f    # 0x1, spec -1
        .byte   0
        .byte   0       # 0x3, cleanup
        .byte   0x7d    # to 0x1
        .byte   0x3     # 0x5, type 3
        .byte   0x7d    # to 0x3
        .byte   0x4     # 0x7, type 4
        .byte   0x7d    # to 0x5
        .byte   0x4     # 0x9, type 4
        .byte   0x77    # to 0x1
        .byte   0x3     # 0xb, type 3
        .byte   0x7d    # to 0x9
        .byte   0x7c    # 0xd, spec -4
        .byte   0x75    # to 0xb
        .align  2 # type table
        .ehword         _ZTIPc          # 5 (char *)
        .ehword         _ZTIi   # 4 (int)
        .ehword         _ZTIf   # 3 (float)
        .ehword         _ZTIPi# 2 (int *)
        .ehword         _ZTIPv          # 1 (void *)
$LLSDATT1:          # Exception specification table
        .byte   0x1     # -1
        .byte   0x2
        .byte   0
        .byte   0x5     # -4
```

```
.byte   0x4
.byte   0
```

This is approximately 83 bytes of data. The region start & landing pad encodings are likely to be 2 bytes in a more larger realistic example, which would bring it to around 100 bytes.

7.3 Implications of DWARF Implementations

The DWARF scheme has a number of implications to implementations, and diverging from these will complicate any implementation of an alternative scheme.

- The exception regions described do not nest. Unwinding will match exactly one exception region, and that region's actions must describe how any possible exception for that region should be handled. The single landing pad for that region handles all such exceptions.
- Code gaps between exception regions may propagate exceptions. These need to be detected and cause termination in a language-specific manner. However, often the gaps contain no exception raising code, or are zero-length.
- The type table provides function-wide ordinals for each type of thrown object.
- The exception specification indices provide function-wide ordinals for each distinct exception specification.
- The action table threading provides the necessary nesting information. Actions for an exception region are ordered such that the innermost exception possibility is handled first.
- Landing pad code handles the exceptions specific to that landing pad and hands off other exceptions to its successor landing pad.
- Regions that require no handling by the function, but through which unwinding should continue are explicitly noted.

The following observations are made, which guide the design of the compact LSDA encoding:

- Exception code ranges are sorted and must be linearly searched. Therefore it would be more compact to specify each relative to the previous one, rather than relative to a fixed base.
- The landing pad is often close to the exception region that uses it. Therefore it is better to use the end of the exception region as the reference point, than use the function base address.
- The action table can be integrated directly with the exception region definition itself. This removes one indirection. The threading of actions can still occur, by providing an offset to the next exception encoding of interest.
- Often the action threading is to the next exception region, so optimizing that case is important.
- Catch types and exception specification type lists cannot easily be encoded inline with the exception regions themselves. It is necessary to preserve the unique indices that are automatically created by the DWARF scheme.

8 Dynamic Object Initialization and Finalization

In order to support mixing DWARF and Compact exception handling at the shared object level, it is

necessary to extend the existing DWARF scheme used to locate the shared library containing a PC value of interest.

8.1 Existing Behaviour

The GNU toolchain essentially supports two mechanisms for locating the set of FDEs to search.

- A legacy scheme using constructor and destructor functions.
- A new header-based scheme using a dedicated ELF segment.

8.1.1 Legacy Scheme

Calls to __register_frame_info_bases are inserted into crtbegin/crtend, which register and deregister the table for the current shared object or executable. The registration function is passed the address of the .eh_frame section, a unique static object and the (dynamic) address of the data segment. It initializes the object and links it into a chain of such objects.

At unload time, the deregistration function unlinks the unique object.

This scheme is suitable for all targets that have the features to support C++.

8.1.2 ELF Segment Scheme

This scheme relies on the static linker to generate an EH Frame Header. The GNU linker's –eh-frame-hdr option enables this, and it is passed by default for Linux targets. The frame header consists of a 1 byte version number, some format data, a count of the FDEs, and then a trailing table of sorted {PC, FDE pointer} tuples.

The EH Frame Header is placed in a .eh_frame_hdr section and PT_GNU_EH_FRAME segment, and is 4-byte aligned. The unwinder can locate it by using dl_iterate_phdr.

8.2 Compact EH Frame Header

The compact encoding extends the ELF Segment scheme, by using a new format EH Frame Header. It is not compatible with the legacy scheme.

The compact EH Frame Header has the following format:

- A 1 byte version number. We shall pick 2.
- A 1 byte encoding of the R_MIPS_EH relocation interpretation (see Section 11). This is a DW_EH_PE_xxx value and may be DW_EH_PE_pcrel | DW_EH_PE_data4, DW_EH_PE_datarel | DW_EH_PE_data4 | DW_EH_PE_indirect.
- 2 bytes of alignment padding, initialized to 0
- A uint32 encoding of the number of Index Table entries.

The EH Frame Header is immediately followed by the Index Table (Section 9)

The compact scheme is compatible with the DWARF scheme, as the version number distinguishes the two.

9 Index Tables

The compact scheme generates index table fragments during assembly, and constructs the sorted

table during linking.

In common with the ARM EABI, and GNU EH Frame Header schemes, the index table consists of pairs of self-relative pointers to function base addresses and language independent unwinding data. If the unwinding data is sufficiently small, it can be placed inline, instead of via indirection. The least significant bit of the function base field is set for inline unwinding data.

For out-of-line unwinding data, the second entry contains a pointer, aligned to a 2-byte boundary. Bit-0 is set to indicate whether the out-of-line information is DWARF-based, and clear to indicate the compact scheme is used. In this way the DWARF-based scheme can be intermixed with the compact scheme, in newly generated code. This may be necessary for functionality beyond that required for C++ exception handling (such as non-call exceptions and asynchronous unwinding). By allowing DWARF unwinding to still be generated, we know that the functionality offered with the compact scheme cannot regress – although, of course, the compactness may be degraded.

Format	Data Representation		Notes
Inline compact	Function offset	0	Unwind data is an array of 4 chars, regardless of endianness.
	Compact unwind data		
Out-of-line compact	Function offset	1	
	Compact unwind offset	0	
Legacy	Function offset	1	
	Legacy unwind offset	1	

All offsets are R_MIPS_REL32 relocations, with an addend of 0, 1 or -1[1] to set bit 0 appropriately.

Because the index table describes half-open intervals, there is the possibility of unwinder crashes, if code lacking unwind information is placed after unwindable code, but nevertheless is found during unwinding. In a correct application, this should not occur. However, in a robust system such a situation should be detected and result in a diagnosable error, rather than uncontrolled behaviour.

To prevent this, any section for which an index table entry is emitted, must also emit a terminating index table entry indicating subsequent code is non-unwindable. If, in the final link, the next input section has unwind information, an index table entry covering zero bytes will result. The static linker must remove such zero-length entries to ensure that all index table PC values are distinct.

Index table fragments shall be emitted to a section named .eh_frame_entry, possibly with additional trailing characters. If an index table fragment is emitted for unwind information in a COMDAT section, the table fragment must be in the same COMDAT group as the unwind information.

Note that shared objects are restricted to 2GB text segment size, even with a 64-bit ABI, thus 32-bit index table offsets are sufficient.

9.1 Linker Behaviour

The static linker must concatenate all input .eh_frame_entry sections, but arrange for them to be

1 A -1 addend would be needed to adjust a MIPS16 or microMIPS function address if a zero LSB is required. Fortunately the code generator will know at this point whether a MIPS16, MIPS32 or microMIPS function address is being specified.

sorted in ascending target function address. This may also involve removing entries for zero-byte spans. It may further optimize the table by eliding adjacent identical entries.

An output .eh_frame_hdr section is generated by creatingan EH Frame Header and placing the concatenated .eh_frame_entry input sections afterwards. Also a PT_GNU_EH_FRAME program segment is created for the .eh_frame_hdr section. Note the output segment would usually also be part of the loadable text segment. It is an error for input object files to contain non-empty .eh_frame_hdr sections.

The global hidden symbol __GNU_EH_FRAME_HDR should also be initialized to the location of the .eh_frame_hdr section, so that bare metal and other non-dynamic environments may locate the index table.

9.2 Searching

How the unwind information for a particular PC is located depends on the target environment

9.2.1 Static Environments

In a static environment the runtime only has to scan a single index table. The address of the EH Frame Header will be the __EH_FRAME_HDR symbol. Runtime code can read the header and then perform a binary search on the trailing index table to locate the entry of interest.

9.2.2 Dynamic Environments

In a dynamic environment, multiple index tables exist. The current ELF Segment scheme uses dl_iterate_phdr to scan each loaded object for its PT_GNU_EH_FRAME segment. This will need augmenting to select the compact scheme as the EH Frame Header version number indicates.

Again, a binary search of the index table will locate the index table entry of interest.

10 Language independent unwinding

With a MIPS-specific scheme, there is little information in a CIE that could not be hard-coded in the specification itself. Indeed, only the personality routine is object-dependent. As such, there is no need for the kind of FDE/CIE separation involved in the DWARF scheme. Also, rather than refer to the LSDA from the FDE, the LSDA can be placed directly after the language independent data. The unwind data is conceptually an array of chars, which is decoded in order of increasing index. The unwind data is 2-byte aligned, so that bit-0 is always clear, and hence need not be encoded in the index table.
The unwind data begins with a generic or builtin header of the following forms:

Format	Data	Notes
Generic compact	000000 | 00 Personality routine offset	PR offset is encoded as an R_MIPS_EH relocation.
Builtin compact	yyyyyy | xx	Builtin PR xx. xx != 00. Remaining bits for PR-specific use.

The upper 6 bits of the first byte are reserved and must be zero. These may be used to extend the range of the personality function version number if needed in the future. The lower 2 bits indicate the personality routine to use. When zero, they indicate the PR routine is specified explicitly. A non-zero value indicates a builtin personality, named '__gnu_compact_prX', where X is a digit 1 to 3. Following the header is a variable length sequence of frame unwind instructions. After the unwind instructions is the personality routine data.
An index table entry using inline compact encoding, must use builtin compact routine 1 and is limited to 3 additional bytes of PR data.
Compact unwind data must be emitted to a section named .gnu_extab, possibly with additional trailing characters. Unwind information describing a COMDAT function must be emitted to athe same COMDAT group as the function (and it's associated index table entry). Legacy DWARF format unwind data will be emitted to .eh_frame and .gcc_except_table in the usual manner.

10.1 Linker Behaviour

The static linker merely needs to concatenate the input .gnu_extab sections in the usual manner. There is no need to place the output in a unique section, although that is encouraged. The output should be placed in the text segment.

If the linker optimizes the index table by eliding duplicate adjacent entries, it should remove the unwind entry associated with the elided entries. Note, only entries using compact builtin PR 0 can be so merged, because the other routines need to know the function start address. It may also choose to optimize the unwind table by eliding identical non-adjacent entries and adjusting the index table entries as appropriate.

10.2 Frame Unwind Data

As described in Section 6, frame unwinding manipulates a virtual register set.

The frame unwind instructions are designed with the following code-generation assumptions:

- If a function has a frame pointer, this register is initialized in the prologue and remains unchanged for the remainder of the function. This implies that the frame pointer for unwindable functions must be a callee-saved register.

  A potential optimization of knowing which caller-saved registers are not used by a particular set of function calls would allow allocating a frame pointer to a non-callee-saved register. However, in order to do that, the complete call graph of the functions in question must be known. At that point, either the functions will be known not to throw exceptions – so unwinding is unnecessary – or it will be known that exceptions may be thrown – so there is no unclobbered caller-saved register available. In both cases we do not need to emit unwind information for a non-callee-saved frame pointer.

- Callee-saved registers are saved in descending register order at descending stack frame addresses.
- MIPS16 or microMIPS use the store multiple instructions available.
- MIPS16 code that needs only one callee-save will use $16 and not $17.
- The first callee-saved register is usually stored just below the CFA.
- The floating point registers are stored below the integer registers.

The DWARF and ARM EABI schemes differ in their approaches to describing unwinding. The DWARF scheme describes where the registers are saved relative to the CFA, and this implicitly describes how to restore state to the previous frame. The ARM scheme directly encodes the actions necessary to restore state. This scheme follows the DWARF approach and instructions fall into one of three categories:

- Those that specify how to locate the CFA, (and the restored SP value)
- Those that specify how other registers were saved, relative to the CFA
- Stop codes.

Because the MIPS ISAs separate adjustment of the stack pointer from the saving of callee-saved registers, we introduce an additional virtual register, the cursor, which indicates the current save point independent of the value of the virtual stack pointer or the CFA. A push operation will decrement the cursor appropriately and record the decremented value as the location of the pushed register.

The frame unwinding instructions are:

Bit pattern	Action	Meaning
00xxxxxx	VR[29] += xxxxxx * ALIGN + ALIGN; cursor = VR[29]	Increment VR[29] by [0x10,0x400) or [0x08,0x200) depending on ABI.
01000xxx,	Pushed VR[16] to VR[16+xxx] & VR[31];	Save callee-save registers and return address.
01001xxx	Pushed VR[16] to VR[16+xxx] , VR[30] & VR[31];	
01010xxx	VR[29] = VR[16 + xxx]; cursor = VR[29]	Restore VR[29] from a frame pointer
01011000 uleb128	VR[29] += (uleb128 + 129) * ALIGN	Large stack adjustment
01011001 xxxxxyyy	Pushed VR[xxxxx] to VR[xxxxx + yyy]	Pushed contiguous integer regs
01011010 xxxxxyyy	Pushed VRF[xxxxx] to VRF[xxxxx + yyy]	Pushed contiguous float regs
01011011	Restore stack pointer	Restore the stack pointer from the CFA
01011100	Finish	PC = VR[31]
01011101	No unwind	
01011110	VR[29] = VR[30]; cursor = VR[29]	Restore VR[29] from a frame pointer
01011111		Spare
01100xxx	Pushed VRF[20] to VRF[20+xxx]	Save callee-save floating point
011010xx	Pushed VRF[20] to VRF[28+xx]	
011011xx	Pushed VR[16],VR[17],VR[18+xx]-VR[23],VR[31]	Pushed callee-save registers as-if by mips16 save multiple.
01110xxx	Pushed VR[16] to VR[16+xxx], VR[28] & VR[31];	n64 ABI – Save callee-save registers, frame pointer and return address
01111xxx	Pushed VR[16] to VR[16+xxx], VR[28], VR[30] and VR[31];	
1xxxxxxx		Spare

Notes:

- ALIGN is either 16 or 8, depending on the ABI. This does not further break ABI-intercompatibility, because they are already incompatible at this level.
- For stack adjustments in the range [65*ALIGN,128*ALIGN] use the 00xxxxxx code twice.
- Codes 01011001 and 01011010 are for unorthodox stack frames. They are not expected to be needed in normal code.
- For MIPS16 saves of $16 or $16 and $17 use the appropriate 01000xxx code.

- For inline unwind data, there is an implicit finish code after the 3 explicit bytes. (Out-of-line unwind data always has an explicit finish code.)
- A function that cannot be unwound should use the inline compact encoding with a single no-unwind code in the data area. The header and first byte will be 0x01, 0x5d. In order to maintain endianness impartiality, the palindromic code 0x015d5d01 must be used so that runtime can check this value simply as a 32-bit code in an endian-neutral manner, if it so chooses.
- Unused trailing bytes of inline data should otherwise be zero.
- Floating point registers explicitly name both parts of an even/odd pair, even if a single instruction would be used to save and restore them. Code generators already need to know how to save and restore these registers in prologues and epilogues.
- A function with no exception regions (i.e., it is simply unwound), can usually be described with 2 or 3 instructions, not counting the finish code. Thus these can be placed inline in the index table.
- Much of the opcode space is unused. This may be allocated later to support non-call exceptions or asynchronous unwinding. The spare bits in the header may be used to encode version information.

10.2.1 Examples

The following simple prologue:

```
addiu  $sp,$sp,-56
sw     $31,52($sp)
sw     $22,48($sp)
sw     $21,44($sp)
sw     $20,40($sp)
sw     $19,36($sp)
sw     $18,32($sp)
sw     $17,28($sp)
sw     $16,24($sp)
```

would be encoded as:

```
00000110    VR[29] += 56
01000110    Pushed VR[16]..VR[22],VR[31]
01011100    Finish
```

Those three opcodes can be placed inline in the index table.

The following prologue in an alloca-using function:

```
addiu  $sp,$sp,-48
sw     $31,44($sp)
```

```
sw      $fp,40($sp)
sw      $20,36($sp)
move    $fp,$sp
sw      $19,32($sp)
sw      $18,28($sp)
sw      $17,24($sp)
sw      $16,20($sp)
```

would have this encoding:

```
01011110    VR[29] = VR[30]
00000101    VR[29] += 48
01001101    Pushed VR[16]..VR[20],VR[30],VR[31]
01011100    Finish
```

As the final opcode is 'Finish', this can be omitted and the remaining 3 placed inline in the index table. If it is placed out-of-line, then the 'Finish' opcode must be present.

10.3 Builtin PR 1

Builtin PR1 indicates there is no language-dependent data. I.E. only frame unwinding information is present.

10.4 Builtin PR 2

Builtin PR 2 is used for C and C++ code that contain exception regions. Refer to Section 7.3 for rationale influencing this encoding. The upper bits of the builtin compact unwind header have the following meanings:

2-5 – Reserved must be zero. May be used for versioning of the PR routine.

6 – If set, exception specification tables precede the type table.

7 – Reserved, must be zero.

The language-specific data is partitioned as follows:

- A uleb128 offset to the end of the region data
- The region data
- Exception specification tables, if present.
- Alignment padding
- Array of type data

The exception specification tables are encoded using a leading uleb128 byte count. If this byte count is zero, it itself is the single exception table (corresponding to an empty 'throw ()' exception specification). Otherwise it indicates the number of following bytes to skip to reach the alignment before the type data. It is not a requirement that a single 'throw ()' exception specification be encoded in this manner, although clearly that is optimal.

The region data consist of:

- A uleb128 region length. Bit zero is set for a no-throw region, and there are no additional fields (landing pad, action or chain), for that case. If the region length is zero, the landing pad field is elided (but action-chain fields are still present). See Section 10.4.4 for an example.
- A sleb128 landing pad offset, relative to the end of the region. A relative value of zero is permitted, and indicates the landing pad immediately follows the region being described. A relative value of -1 indicates an unwind continue region. There is no following action-chain field in that case.
- A sleb128 action and chain value.
- A sleb128 action extension value, if needed.

Bit 0 is also used to indicate the ISA (mips16/micromips vs MIPS32/MIPS64). However, as the ISA cannot change except at function boundaries, it is not necessary to encode this explicitly in length and displacement values.

The action-chain value is partitioned with bits 0...1 holding a signed action value and the remaining bits holding the signed chain value. If the action value is -2, an extension action value is present. This is a sleb128 encoded action value.The action values are:

- = 0 – a cleanup.
- > 0 – a catch. The type is indicated by the indexed slot in the type table (which is 1-based).
- < 0 – an exception specification. The negated value is a 1-based offset into the exception specification table data.

Note, these action values have the same interpretation as the DWARF scheme. Action values -1, 0 and 1 can be encoded either within the combined action-chain sleb128, or as an extension action sleb128.

The type table is indexed by 1-based values. A table slot holding an uninterpreted value of zero is

used to indicate any type, which is needed for a 'catch (...)' clause.

The exception specification data consists of non-zero values used to index the type table, and terminating zero values. An empty exception specification is therefore a single zero byte. Such a zero byte can be overlaid onto the end of a non-empty exception specification, but this is not an encoding requirement.

The chain value specifies how to locate the next action of interest, should the current action not terminate processing. A value of zero indicates the end of the chain, and unwinding should continue in the caller's context in that case. The chain displacement does not count no-throw or unwind-continue regions. Negative values are permissible, and can be handled by rescanning the action records from the beginning.

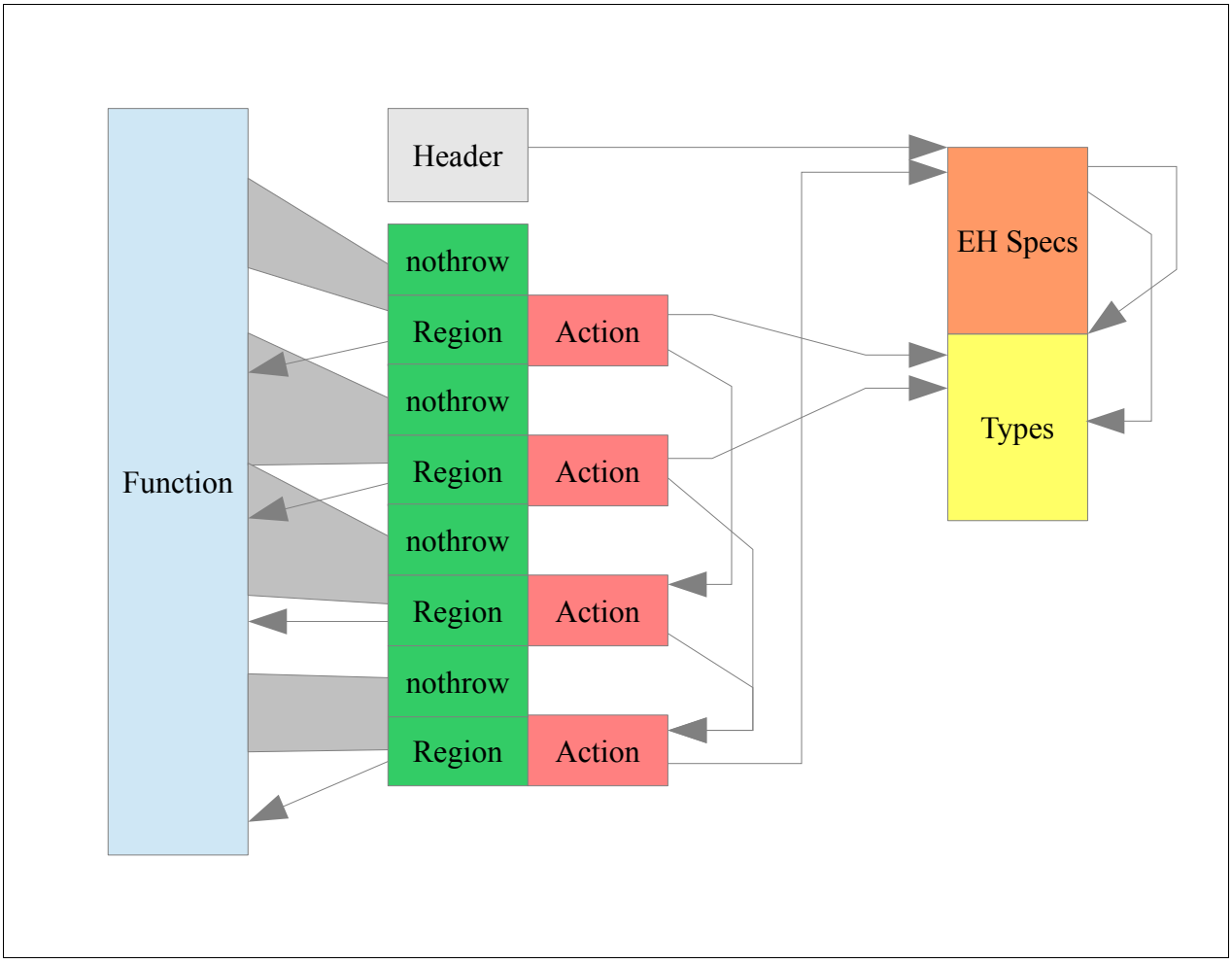
It may be necessary to insert dummy zero-length regions to encode action chain values that do not commence a chain in their own right. Such regions have no landing pad. For instance, if the only catch clauses always encode the same set of types. See Section 10.4.4 for an example.

As the region locations are only described by length values, it is desirable to grow the exception regions over non-throwing code, so that they abut. If regions cannot abut – for instance it is not provable that the intermediate code does not contain exceptions, explicit no-throw regions must be inserted. Such no-throw regions are a single uleb128 encoding, and therefore cannot be a less dense encoding than the DWARF scheme, which requires this on every region.

10.4.1 Data Layout

The following diagram shows how these tables are laid out. Compare with the DWARF diagram in Section 6.

- All of the pointers are relative offsets. The base of the offset is some other defined point in the data structure, or from the offset itself (i.e. Self-relative).
- NULL entries are an appropriate zero.
- The EH spec table (orange) has a header that indicates the offset to the padding before the type table.
- The type table (yellow) is aligned to a 4 byte boundary.
- The type table is in ascending order indexed from the lowest address – rather than DWARF's descending order indexed from the highest address.
- The region entries (green) and action entries (red) are intermixed. There is no longer an explicit offset from each region entry to a specific action entry. Obviously this means that the initial action entry cannot be shared, but that is not a common occurence
- A header (grey) points to the EH spec table. This header abuts the region table, and the region table abuts the EH spec table.
- The EH spec table points to entries in the type table. The end of an EH spec sequence is indicated by a NULL entry.
- Each region table entry indicates a code length. Thus regions always abut. In the diagram, additional no-throw regions have been placed before each active region, and the grey polygons show the effective code ranges. Thus in this case, there is a simple mapping between region numbers in the DWARF scheme and the compact scheme.
- Each non-nothrow non-zero-sized region table entry is either an unwind-propagation region, or an action-processing region. The encoded landing-pad value indicates which. specifies The landing pad is specified by offset from the end of the region's code. This landing pad is responsible for processing any action for exceptions emanating from the region's code range.
- Each action-processing region table entry contains an action entry.
- Each action entry points to a successor region. This is a count of action entries to advance (or retard). The end of the chain indicates the exception is unhandled and unwinding should continue in the caller's context.
- Each action table entry points to either
    - a specific type (to catch that type). A specific type can be shared between multiple action table entries (but usually are not).
    - an EH spec, to check an EH specification. A specific EH spec can be shared, but usually is not.
    - or is NULL to indicate a cleanup.
- If the action matches, the original region's landing pad is called. Information about what matched is passed to the landing pad.
- If the action fails to match, the chain is followed and the next action is checked.



10.4.2 Coding Example

The C++ example from Section 7.2, the compact scheme with the above layout encodes the region data as:

```
.byte 0x42                          # PR2, eh-specs
.uleb128 $LLSDACSE1-$LLSDACSB1      # call site length
$LLSDACSB1:
.uleb128 $LEHB0 - $LFB1 + 1         # region 0, no-throw gap
.uleb128 $LEHE0 - $LEHB0                    # region 1[0]2 length
.sleb128 $L26 - $LEHE0              # landing pad
.sleb128 (2<<2) | 2                 # action-ext, chain +1(R3)
.sleb128 4                          # catch 4 (int)
.uleb128 $LEHB1 - $LEHE0 + 1        # region 2, no-throw gap
.uleb128 $LEHE1 - $LEHB1                    # region 3[1] length
.sleb128 $L27-$LEHE1                # landing pad
```

2 Numbers in [] are the original DWARF region number

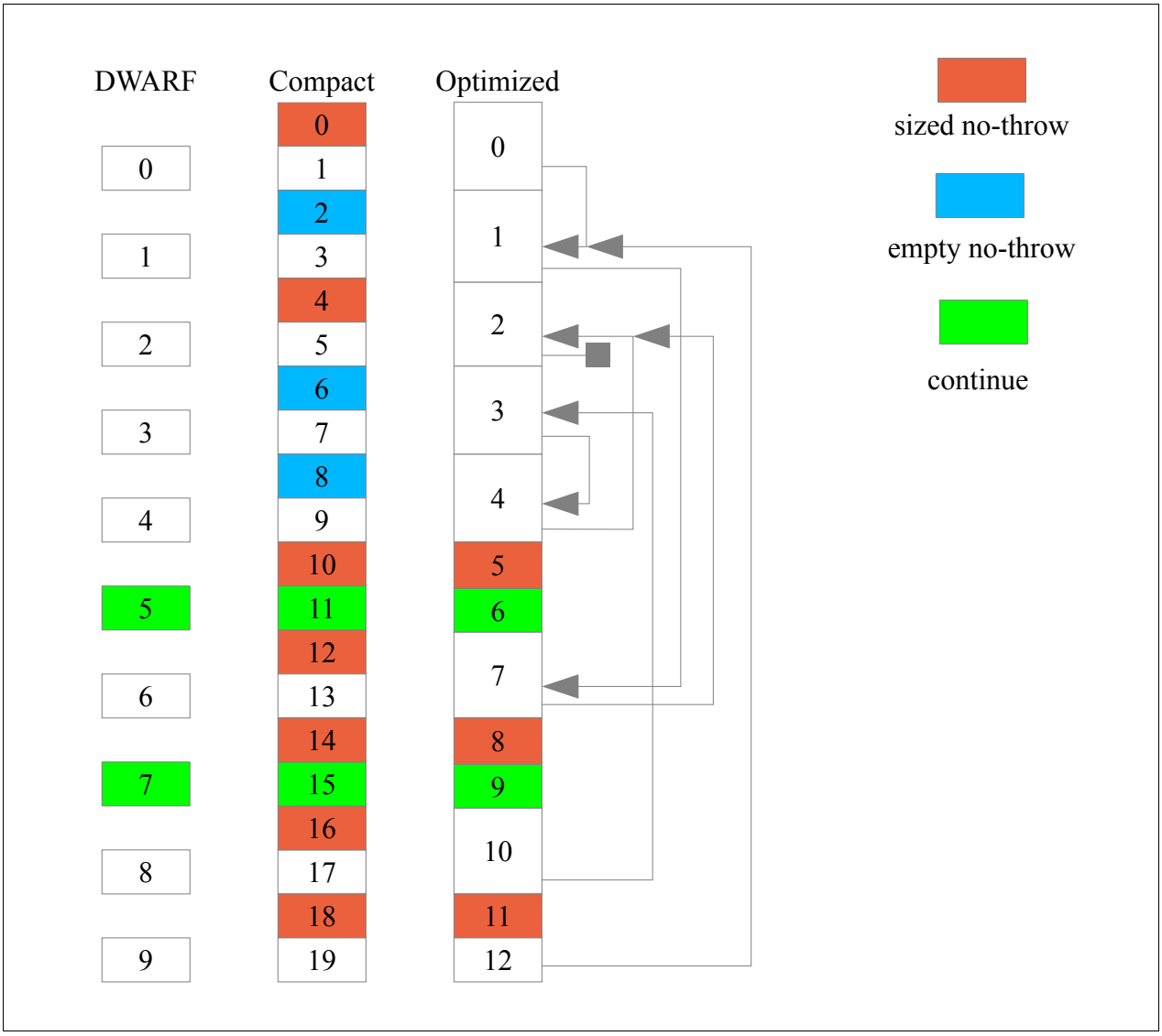
```
.sleb128 (10 << 2) | 2          # action-ext, chain +4(R13)[3]
.sleb128 3                      # catch 3 (float)
.uleb128 $LEHB2 - $LEHE1 + 1    # region 4, no-throw gap
.uleb128 $LEHE2 - $LEHB2            # region 5[2] length
.sleb128 $L29 - $LEHE2          # landing pad
.sleb128 3                      # eh-spec -1, chain end
.uleb128 $LEHB3 - $LEHE2 + 1    # region 6, no-throw gap
.uleb128 $LEHE3 - $LEHB3            # region 7[3] length
.sleb128 $L30 - $LEHE3          # landing pad
.sleb128 (2<<2) | 2             # action-ext, chain +1(R9)
.sleb128 3                      # catch 3 (float)
.uleb128 $LEHB4 - $LEHE3 + 1    # region 8, no-throw gap
.uleb128 $LEHE4 - $LEHB4            # region 9[4] length
.sleb128 $L31 - $LEHE4          # landing pad
.sleb128 (-4<<2) | 2          # action-ext, chain -2(R5)
.sleb128 4                      # catch 4 (int)
.uleb128 $LEHB5 - $LEHE4 + 1    # region 10, no-throw gap
.uleb128 $LEHE5 - $LEHB5        # region 11[5]
.sleb128 -1                     # unwind-continue
.uleb128 $LEHB6 - $LEHE5 + 1    # region 12, no-throw gap
.uleb128 $LEHE6 - $LEHB6            # region 13[6] length
.sleb128 $L32 - $LEHE6          # landing pad
.sleb128 (-8 << 2)              # cleanup, chain -3(R5)
.uleb128 $LEHB7 - $LEHE6 + 1    # region 14, no-throw gap
.uleb128 $LEHE7 - $LEHB7 + 1    # region 15[7]
.sleb128 -1                     # unwind-continue
.uleb128 $LEHB8 - $LEHE7 + 1    # region 16, no-throw gap
.uleb128 $LEHE8 - $LEHB8            # region 17[8] length
.sleb128 $L33 - $LEHE8          # landing pad
.sleb128 (-10<<2) | 7           # action-ext, chain -3(R7)
.sleb128 -4                     # eh-spec -4
.uleb128 $LEHB9 - $LEHE8 + 1    # region 18, no-throw gap
```

3 +6(R19) would also be an acceptable chain value.

```
.uleb128 $LEHE9 - $LEHB9                    # region 19[9] length
.sleb128 $L28 - $LEHE9                      # landing pad
.sleb128 (-16<<2)                           # cleanup, chain -6(R3)
$LLDACSE1:
.uleb128 $LLSDATT1 - $LLSDATTD1             # eh-spec length
$LLSDATTD1:
.byte 0x1, 0x2, 0                           # eh-spec -1 (void *, int *)
.byte 0x5, 0x4, 0                           # eh-spec -4 (char *, int)
$LLSTATT1:
.align 2                    # type table
.ehword     _ZTIPv          # type 1(void *)
.ehword     _ZTIPi# type 2 (int *)
.ehword     _ZTIf  # type 3 (float)
.ehword     _ZTIi  # type 4 (int)
.ehword     _ZTIPc          # type 5 (char *)
```

This example has methodically added explicit no-throw regions before each DWARF-specified region. The encoding is around 70 bytes, approximately 16% to 30% smaller than the DWARF encoding.

The following diagram shows the mapping between DWARF region numbering, unoptimized compact numbering and optimized compact numbering:



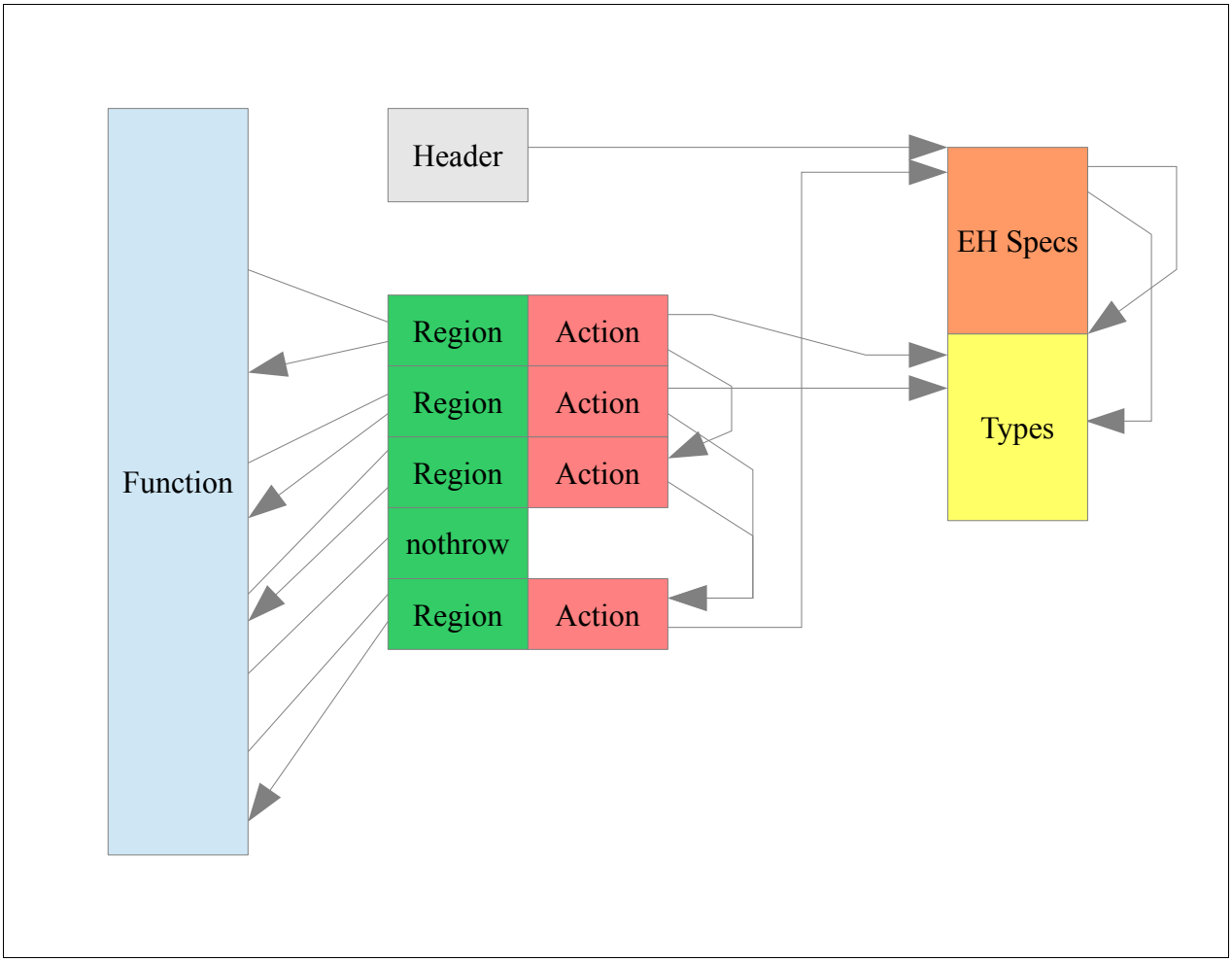
Section 10.4.3 discusses optimizing this layout.

10.4.3 Optimized Data Layout

An optimization to this layout is to reduce the number of no-throw regions. This can be achieved by:

-
- Omitting zero-sized no-throw gaps.
- Growing exception regions across instructions that cannot raise an exception (i.e. over non-call instructions). Growing regions in this manner frequently results in abutting exception regions.
- If it is assumed that exceptions are never raised in no-throw regions (for instance similar to an optimization that removes exception-specification checking), all no-throw regions can be coalesced with an adjacent region, regardless of that region's semantics.

Once no-throw regions are eliminated in this way, there is no longer a simple region number mapping between DWARF and compact schemes.
The following diagram shows such an optimization:



Optimized Coding Example
The example in Section 10.4.2 can be optimized to:

```
1.  .byte 0x42                          # PR2, eh-specs
    .uleb128 $LLSDACSE1-$LLSDACSB1      # call site length
    $LLSDACSB1:
    .uleb128 $LEHE0 - $LFB1             # region 0[0] length
    .sleb128 $L26 - $LEHE0              # landing pad
    .sleb128 (1<<2) | 2                 # action-ext, chain +1
    .sleb128 4                          # catch 4 (int)
    .uleb128 $LEHE1 - $LEHE0                    # region 1[1] length
    .sleb128 $L27-$LEHE1                # landing pad
    .sleb128 (4<<2) | 2                 # action-ext, chain +4
    .sleb128 3                          # catch 3 (float)
    .uleb128 $LEHE2 - $LEHE1                    # region 2[2] length
    .sleb128 $L29 - $LEHE2              # landing pad
```

```
.sleb128 (0<<2) | 3                     # eh-spec -1, chain end
.uleb128 $LEHE3 - $LEHE2                        # region 3[3] length
.sleb128 $L30 - $LEHE3                  # landing pad
.sleb128 (1<<2) | 2                     # action-ext, chain +1
.sleb128 3                              # catch 3 (float)
.uleb128 $LEHE4 - $LEHE3                        # region 4[4] length
.sleb128 $L31 - $LEHE4                  # landing pad
.sleb128 (-2<<2) | 2                # action-ext, chain -2
.sleb128 4                              # catch 4 (int)
.uleb128 $LEHB5 - $LEHE4 + 1            # region 5, no throw
.uleb128 $LEHE5 - $LEHB5                        # region 6[5] length
.sleb128 -1                             # unwind-continue
.uleb128 $LEHE6 -  $LEHE5               # region 7[6] length
.sleb128 $L32 - $LEHE6                  # landing pad
.sleb128 (-3<<2) | 0                # cleanup, chain -3
.uleb128 $LEHB7 - $LEHE6 + 1            # region 8, no-throw
.uleb128 $LEHE7 - $LEHB7                                # region 9[7]
.sleb128 -1                             # unwind-continue
.uleb128 $LEHE8 -  $LEHE7               # region 10[8] length
.sleb128 $L33 - $LEHE8                  # landing pad
.sleb128 (-2<<2) | 2                # action-ext, chain -2
.sleb128 -4                             # eh-spec -4
.uleb128 $LEHB9 - $LEHE8 + 1            # region 11 no-throw
.uleb128 $LEHE9 - $LEHB9                        # region 12[9] length
.sleb128 $L28-$LEHE9                    # landing pad
.sleb128 (-6<<2)|0                      # cleanup, chain -6
$LLDACSE1:
.uleb128 $LLSDATT1 - $LLSDATTD1     # eh-spec length
$LLSDATTD1:
.byte 0x1, 0x2, 0                       # eh-spec -1 (void *, int *)
.byte 0x5, 0x4, 0                       # eh-spec -4 (char *, int)
$LLSTATT1:
.align 2                            # type table
```

```
        .ehword     _ZTIPv              # type 1 (void *)
        .ehword     _ZTIPi          # type 2 (int *)
        .ehword     _ZTIf           # type 3 (float)
        .ehword     _ZTIi           # type 4 (int)
        .ehword     _ZTIPc              # type 5 (char *)
```

This example does not presume exceptions are never raised in no-throw regions, and thus preserves the minium number of no-throw regions required for language conformance.

This example grows some eh regions to abut and merges adjacent no-throw regions. Note that the chain displacement values are unchanged from the unoptimized example, as the number of non-no-throw regions is unchanged.

The encoding is around 62 bytes, 25%-40% smaller than the DWARF encoding.

10.4.4 Multiple Catch Clauses

As mentioned above, if a function contains a single series of catch clauses for a single try, there can be DWARF actions that are not directly reachable from an exception region. When converting to the Compact scheme, dummy zero length regions need to be inserted to own those actions. An example would be:

```
void Baz () {
  try {
    Foo ();             // Might throw
  } catch (char) {
    BarChar ();         // A no-throw function
  } catch (int) {
    BarInt ();          // A no-throw function
  } catch (float) {
    BarFlt ();          // A no-throw function
  }
}
```

The following DWARF data is generated:

```
.LLSDACSB2:         # Call-site table
.uleb128 .LEHB0-.LFB2    # region 0 start
.uleb128 .LEHE0-.LEHB0  # length
.uleb128 .L8-.LFB2          # landing pad
.uleb128 0x5                # action 5
.uleb128 .LEHB1-.LFB2    # region 1 start
```

```
.uleb128 .LEHE1-.LEHB1  # length
.uleb128 0x0            # no throw
.uleb128 0x0            # nothing

.LLSDACSE2:     # Action record table
.byte   0x1     # action 1, catch float
.byte   0x0     # end of chain
.byte   0x2     # action 3, catch int
.byte   0x7d # goto action 1
.byte   0x3     # action 5, catch char
.byte   0x7d # goto action 3

.align 4        # Type table
.long _ZTIc     # type 3, char
.long   _ZTIi   # type 2, int
.long   _ZTIf   # type 1, float
.LLSDATT2:
```

As can be seen, no region refers directly to actions 1 or 3. A possible Compact encoding is:[4]

```
.byte 0x2                                   # PR2
.uleb128 $LLSDACSE2-$LLSDACSB2  # call site length
$LLSDACSB2:
.uleb128 $LEHB0 - $LFB1 + 1             # region 0 no-throw gap
.uleb128 $LEHE0 - $LEHB0                    # region 1[0] length
.sleb128 $L7 - $LEHE0                   # landing pad
.sleb128 (1<<2) | 2                     # action-ext, chain +1
.sleb128 3                              # catch 3 (char)
.uleb128 $LEHB1 - $LEHE0 + 1            # region 2 no-throw gap
.uleb128 $LEHE1 - $LEHB1 + 1            # region 3[1] no throw gap
.uleb128 0                              # region 4, dummy
.sleb128 (1<<2) | 2                     # action-ext, chain +1
.sleb128 2                              # catch 2 (int)
.uleb128 0                              # region 5 dummy
.sleb128 1                              # catch 1 (float), no chain
```

4 This example does not perform any of the region coalescing optimizations described in Section 10.4.3.

```
$LLDACSE2:
.align 4                # Type table
.long   _ZTIf           # type 1 (float)
.long   _ZTIi           # type 2 (int)
.long   _ZTIc           # type 2 (char)
```

This appends two trailing dummy regions to hold those actions. This example is 12 bytes compared to the DWARF scheme's 14 bytes (excluding the type table).

10.4.5 Different Catch Clause Chains

If a function contains a set of catch clauses with different sets of caught type, there can be multiple references to a particular type in different action chains. When converting to the Compact scheme, the distinct chains must be kept separate. An example would be:

```
void Baz () {
  try {
    Foo ();             // Might throw
  } catch (char) {
    BarChar ();         // A no-throw function
  } catch (int) {
    BarInt ();          // A no-throw function
  }

  try {
    Foo ();             // Might throw
  } catch (char) {
    BarChar ();         // A no-throw function
 } catch (int) {
    BarInt ();          // A no-throw function
  } catch (float) {
    BarFlt ();          // A no-throw function
  }
}
```

The following DWARF data is generated:

```
.LLSDACSB2:         # Call-site table
.uleb128 .LEHB0-.LFB2    # region 0 start
.uleb128 .LEHE0-.LEHB0  # length
```

```
.uleb128 .L13-.LFB2             # landing pad
.uleb128 0x3                    # action 3
.uleb128 .LEHB1-.LFB2   # region 1 start
.uleb128 .LEHE1-.LEHB1  # length
.uleb128 .L12-.LFB2             # landing pad
.uleb128 0x9                    # action 9
.uleb128 .LEHB2-.LFB2   # region 2 start
.uleb128 .LEHE2-.LEHB2  # length
.uleb128 0x0                    # no throw
.uleb128 0x0                    # nothing

.LLSDACSE2:         # Action record table
.byte   0x2             # action 1, catch 2 (int)
.byte   0x0             # end
.byte   0x3             # action 3, catch 3 (char)
.byte   0x7d            # goto action 1
.byte   0x1             # action 5, catch 1 (float)
.byte   0x0             # end
.byte   0x2             # action 7, catch 2 (int)
.byte   0x7d            # goto action 5
.byte   0x3             # action 9, catch 3 (char)
.byte   0x7d            # goto action 7

.align 4        # Type table
.long _ZTIc     # type 3, char
.long   _ZTIi   # type 2, int
.long   _ZTIf   # type 1, float
.LLSDATT2:
```

As can be seen, actions 3 and 9 both catch 'char', followed by actions 1 and 7, which each catch 'int', but only action 7 is chained to a catch 'float'. A possible Compact encoding is:[5]

```
.byte 0x2                                       # PR2
.uleb128 $LLSDACSE2-$LLSDACSB2  # call site length
$LLSDACSB2:
.uleb128 $LEHB0 - $LFB1 + 1             # region 0 no-throw gap
```

5 This example does not perform any of the region coalescing optimizations described in Section 10.4.3.

```
.uleb128 $LEHE0 - $LEHB0                # region 1[0] length
.sleb128 $L12 - $LEHE0                  # landing pad
.sleb128 (2<<2) | 2                     # action-ext, chain +2
.sleb128 3                              # catch 3 (char)
.uleb128 $LEHB1 - $LEHE0 + 1            # region 2 no-throw gap
.uleb128 $LEHE1 - $LEHB1                # region 3[1] length
.uleb128 $l7 - $LEHE1                   # landing pad
.sleb128 (2<<2) | 2                     # action-ext, chain +2
.sleb128 3                              # catch 3 (char)
.uleb128 $LEHB2 - $LEHE1 + 1            # region 4 no-throw gap
.uleb128 $LEHE2 - $LEHB2 + 1            # region 5[2] no throw gap
.uleb128 0                              # region 6, dummy
.sleb128 2                              # action-ext, no chain
.sleb128 2                              # catch 2 (int)
.uleb128 0                              # region 7 dummy
.sleb128 2                              # action-ext, chain +1
.sleb128 2                              # catch 2 (int)
.uleb128 0                              # region 8 dummy
.sleb128 1                              # catch 1 (float), no chain

$LLDACSE2:
.align 4            # Type table
.long _ZTIf         # type 1 (float)
.long  _ZTIi        # type 2 (int)
.long  _ZTIc        # type 2 (char)
```

This appends three trailing dummy regions to hold these different action chains. This example is 20 bytes compared to the DWARF scheme's 22 bytes (excluding the type table).

10.5 Builtin PR 3

Builtin PR 3 is the same as PR 2, except landing pad offsets are relative to an explicit base address prepended to the LSDA information. It is an int32 indicating the self-relative offset to the landing pad base address. This is useful for noncontiguous functions where landing pads are placed in a remote section.

Following the LPStart value, the table as is for builtin PR 2.

11 Relocations

A new static relocation, R_MIPS_EH, is defined. The semantics of this relocation depend on whether static or dynamic linking is provided.

A GNU extension using relocation number 249 shall be used. The relocation address need not be naturally aligned.

11.1 Static code

For static code generation, the calculation is the same as an R_MIPS_REL32 relocation.

At runtime, the following expression provides the relocated value, if 'ptr' points to the relocation location.

- (ptrdiff_t)ptr + *(int32_t __attribute__((packed))*)ptr

Note that this limits static executables to be no bigger than 2GB in the 64bit ABIs. However, they may be placed at any base address within the memory space.

11.2 PIC code

For PIC code generation, a 32- or 64-bit GOT-table entry must be allocated to refer to the (dynamically resolved) target address. Once the GOT entry has been allocated, the static calculation is as for an R_MIPS_GPREL32 relocation (except that the calculation uses the GOT slot address). The GOT-slot has an associated R_MIPS_{32,64} dynamic relocation emitted – and that will of course be at a naturally aligned location

At runtime, the following expression provides the relocated value, if 'ptr' points to the relocation location, and 'gp' is the global pointer value:

- *(ptrdiff_t *)((ptrdiff_t)gp + *(int32_t __attribute__((packed)) *)ptr)

The $gp value used must be the one associated with the (primary) GOT of the shared object containing the relocation (the address in 'ptr').

Note that the GOT is restricted to 2GB in the 64bit ABIs. However, the target address may be placed anywhere in the address space.

12 Implications for Implementations

This scheme is sufficiently general, apart from the exact encoding of the register unwind commands, that it could probably be used on other architectures (variable-length ISAs will be problematic). That might make it easier to implement by allowing update of generic structures and files, rather than having to keep the new scheme hidden behind hooks or something.

12.1 GCC

- Unwinding is specified by .cfi directives. Per-function information will need emitting to select the desired encoding scheme.
- Exception tables will need to select the right format on a per-function basis. The basic format of the tables is the same though, so there should not be significant work in the exception machinery.
- A new option will need adding to select the exception scheme.

12.2 GAS

- .cfi directive handling will need augmenting to select between the new and old schemes
- The new scheme should be able to encode the stack frame as described by existing directives. The current directives build an internal representation of the stack frame, which can then be optimized for MIPS.

12.3 LD

- The scripts will need augmenting to place the .eh_frame_hdr and .gnu_extab sections.
- The linker will need augmenting to sort the .eh_frame input sections, attach a compact EH Frame Header and define the __GNU_EH_FRAME_HDR symbol.
- The linker will need augmenting to process R_MIPS_EH relocations.
- It should be an error to encounter .eh_frame input sections without having specified the –eh-frame-hdr option.

12.4 Libgcc

- A new unwind-compact.c file will be needed, probably wrapping pieces of unwind-dw2.c. It should call out to an _Unwind_Find_Index with the following suggested prototype:

```
struct compact_eh_base {
  void *tbase;  // text segment base (as for DWARF)
  void *dbase;   // data segment base (as for DWARF)
  void *func;  // function start (as for DWARF)
  void *entry; // FDE or compact encoding
};
enum compact_entry_type {
  CET_not_found,
  CET_FDE, // an FDE has been found
  CET_inline, // an inline encoding found (entry will point into the index table)
  CET_outline, // an out-of-line encoding
};
enum compact_entry_type UFI (void *pc, struct compact_eh_base *ptr);
```

- unwind-dw2-fde needs augmenting or wrapping in line with Section 8.2.
- An implementation of _Unwind_Find_Index suitable for a static executable will be needed.
- There is no requirement for the complexity of the ARM implementation. That implementation allows some optimization of the _Unwind_Resume path and initial setup. However, the initial setup must remain compatible with the DWARF scheme and the complexity of the Unwind_Resume caching is probably not worthwhile, given the assumption that exception handling is not time-critical.

12.5 EGLIBC

- The unwinder subsystem (libc/sysdeps/generic/unwind-*) will need augmenting to accept version 2 EH Frame Headers, and implement the same compact unwinding as libgcc.
- The pthread subsystem (libc/sysdeps/gnu) wraps the libgcc functions (obtained via dl_open).

It will need augmenting to wrap the new personality routines.

- As with libgcc, there is no need for the complexity of the frameless ARM EABI wrapping.

12.6 uCLibc

- Similar changes as for EGLIBC.

12.7 libsupc++

- New personality routines will need adding. These will be very similar to the ARM personality routines. The two phase unwinding requires recording the location of the caught exception, cleanup etc during phase1, so it may be refound in phase2 via simple pointer comparisons. The exception object has four fields available for recording this, which are sufficient.

13 Table Size

Examining libsupc++, GCC's language support library, built for MIPS32, I find 45 .o files containing 184 functions, 20 of which catch exceptions in some way. There are 184 FDEs and 45 CIEs (a bug in the toolchain emits FDEs for leaf functions, but that is accounted for in this analysis). The .gcc_except_table is 604 bytes, thus about 30 bytes per function (including the dynamically allocated 4 byte index table entry). The .eh_frame is 4788 bytes, giving another 30 bytes per catching function.

Using the compact format, 164 functions should be encodeable using just the 8 bytes of the index table, for a saving of about 75%.

These size estimates cannot be guaranteed, and actual values will depend on the exception nature of the program being compiled. It is clear that the new encoding is denser than the DWARF scheme and promises to provide smaller text segment sizes.

Note that the runtime code in the support library will be larger than before, because it has to handle both DWARF and compact schemes. The existing code is 27K. It is unlikely the new code will be more than twice that in total.