Revised Issue 250408.1: Replace composite locations & DW_OP_overlay with mapping lists

Introduction

This proposal provides a complete replacement for the concept of composite locations as described in DWARF V5 and of composite storage as proposed in DWARF V6 (DW_OP_piece and friends are removed). It also obviates the need for the yet-to-be proposed DW_OP_overlay operation currently under discussion in the GPU group. Further, it supersedes my earlier proposals 230712.2, 230712.3 and 230712.4 (it is actually inspired by a combination of 230712.4 and the pending DW_OP_overlay proposal.)

The proposal may be considered to have three parts:

1. Section 2.6.3.2.1[3.12.2.1] is the core proposal and updates the earlier draft. Without it, the other two parts are moot.
2. Section 2.6.3.2.2[3.12.2.2] is new material that seeks to optimize the size of the operands when constant.
3. Section 2.6.3.2.3[3.12.2.3] is new material that seeks to integrate additional useful liveliness information at minimum additional cost.

Given part 1, either of the other two can be considered independently.

Note: Section numbers refer to Version 5 while the following number in square brackets is the corresponding section in the current V6 working proposal for locations on the stack (Issue 230524.1).

Discussion

In contrast to the composite location formulations in V5 and proposed in V6, this approach does not notionally construct a complete description of an object in some imaginary address space. Rather it describes mappings only for locations of an object that are not in their “normal” locations when, due to optimization, part(s) or the entire object are promoted to alternate locations (typically registers). The intent is that any and all address arithmetic involved in accessing parts of an object are performed in their normal manner in the containing memory address space of an object but an access to the resulting location involves mapping that location as appropriate to its “current” promoted location prior to accessing any contents.

Briefly, the new group of mapping operations all take four operands and push one result on the stack (with one exception). In half of the group, all four operands are on the stack, while in the other half only the first is on the stack and the other three are inline operands that follow the opcode. The operands are:

1. A “source location” to be mapped to a new, or possibly the same, “target location”.
2. The beginning location of a sequence of contiguous “mapped locations” that are to be mapped.
3. The beginning location of a sequence of contiguous “target locations”.
4. The number of locations in each of the two sequences.

The result is defined as follows:

A. If the source location is not in the range of locations defined by the second and fourth operands, then the result is the same as the source location.
B. Otherwise, let I be the zero-based ordinal position of the source location in the mapped location sequence; the result is the corresponding Ith target location in the target location sequence.

The location information for an object is split into two parts:

- The normal DW_AT_location information that gives the location of the object as a whole. If the complete object moves, but is always contained in a single contiguous sequence of locations, then it is completely describable using a traditional location list. This applies generally to all scalar objects but is sometimes true of composite objects as well.
- A new DW_AT_mapping attribute is introduced whose value is also a location list. The “expressions” in this location list, however, consist solely of a sequence of DW_OP_map[*] operations together with their second, third and fourth operands (which are all pushed or all inline). Currently, only one DW_OP_map[*] operation is ever needed in each entry.

It is helpful to observe that the DW_AT_mapping attribute is only needed if and when an object is not a single consecutive sequence of locations in memory.

To illustrate consider this simple example: Let OBJ be an object of type Cmpl which is a struct or record consisting of two single precision floating point components, RE and IM, as in

```
 type Cmpl is struct {
        float RE;
        float IM;
}
```

Further suppose that in a certain part of the program extending from program counter values PC1 through PC2, the second component IM is promoted to register RX.

The value of the DW_AT_location attribute will be just the simple base location for the object as a whole, &OBJ.

The value of the DW_AT_mapping attribute will be a location list which could look like:

```
        DW_LLE_start_end, PC1, PC2+1
        .uleb 2$-1$                     ! Size of mapping expression
1$:     DW_OP_addrx (OBJ+4)             ! Push beginning of O.IM
        DW_OP_regRX                     ! Push beginning of target RX
        DW_OP_const4                    ! Size of each range
        DW_OP_map
2$:     DW_LLE_end_of_list
2$:
```

Note that a source location that is not in the same address space as the mapped locations is simply not mapped; it becomes the result.

The mapping is completely open-ended on both sides of the mapped sequence. This is deliberate and key to simple handling of hidden storage prior to the notional base address of an object, such as vtable pointers, RTTI or the like, that is seldom promoted (but if they are, they can be mapped like any other component). It also works well for C implicit arrays which have no defined length (other than that imposed by the size of the containing address space).

To find the location OBJ.RE, look up OBJ, get its base address from the DW_AT_location attribute, look up component RE in its type, get its offset from the DW_AT_data_member_location attribute and add them together—all exactly as in classic DWARF. Then, push the result &OBJ.RE on the stack and scan the mapping list for a matching entry.

- If the current PC is not in the range PC1..PC2, there is no match and the result on the stack is the same as the input, namely the location of OBJ.RE.
- If the current PC is in the range PC1..PC2 then there is a match for the mapping entry but the source location for &OBJ.RE is not in the range to be mapped so the result is again &OBJ.RE.

To find the location of OBJ.IM, the same process causes &OBJ+4 to be pushed on the stack. Then

- If the current PC is not in the range PC1..PC2, there is no match and the result on the stack is the same as the input, namely the location of OBJ+4.
- If the current PC is in the range PC1..PC2 then there is a match for the mapping entry AND the source location for &OBJ.RE is in the range to be mapped so the result is the location for register RX.

Now, extend this example by adding a third component to the struct/record: an intensity value ITN, which is double precision floating point. Further suppose that The program occupies a sequence of addresses P1 through PC4 where PC1 < PC2 < PC3 < PC4. Finally suppose component RE is always in memory, component IM is promoted to register RX during PC1..PC3 and component INT is promoted to register RY during PC2..PC4.

The mapping list now looks like:

```
        DW_LLE_start_end, PC1, PC3+1
        .uleb 2$-1$                     ! Size of first mapping expression
1$:     DW_OP_addrx (O+4)               ! Push beginning of O.IM
        DW_OP_regRX                     ! Push beginning of target RX
        DW_OP_const4                    ! Size of each range
        DW_OP_map
2$:     DW_LLE_start_end, PC2, PC4+1
        .uleb 4$-3$                     ! Size of second mapping expression
3$:     DW_OP_addrx (O+8)               ! Push beginning of O.ITN
        DW_OP_regRY
        DW_OP_const8
        DW_OP_map
~~4$:~~   DW_LLE_end_of_list
4$:
```

Note that the result of the first DW_OP_map operation becomes the first parameter of the second DW_OP_map operation.

It is worth highlighting ~~here~~ that there is nothing special ~~here~~ about the fact that in the PC range PC2..PC3 ~~even though~~ there are two components that are mapped. Because the memory ranges for the two components do not overlap there is no interaction between the two. Because the memory ranges do not overlap, an implementation can end evaluation of the mapping list as soon as a match is found.

Finally, it is expected that a large percentage of cases, all three of these special cases apply:

1. Component location within the containing object can be easily determined as an offset relative to the base of the object (for example, when all components are of fixed size), where the needed base address is given by the DW_AT_location (or similar) attribute.
2. The target location is a register that can be identified using its ABI defined register number.
3. The size of the mapped and target ranges is ~~fixed~~ the same constant value.

Given these constraints, the last three operands than be expressed as unsigned LEB128 integers that are inline (following) the mapping opcode. These constraints happen to apply in the above example, which can thereby be simplified to the following:

```
        DW_LLE_start_end, PC1, PC3+1
        .uleb 2$-1$                     ! Size of first mapping expression
1$:     DW_OP_mapc
        .uleb 4                         ! Offset of O.IM relative to O
        .uleb <ABI code for register RX>
        .uleb 4                         ! Size of mapped and target ranges
2$:     DW_LLE_start_end, PC2, PC4+1
        .uleb 4$-3$                     ! Size of second mapping expression
3$:     .uleb 8
        .uleb < ABI code for register RY>
        .uleb 8
        DW_OP_mapc
~~4$:~~  DW_LLE_end_of_list
4$:
```

The suffix c at the end of the compact form operator name is intended to suggest "compact".

Now consider how DW_OP_map can be used in a loop unrolling example. Using some unexplained, hand wavy notation and without getting buried in too much detail, consider a bounded location expression appropriate in the body of a loop which is unrolled by a factor two, with the promoted subvector promoted to, say, R8 and R9. The vector VEC being processed is assumed to be a one-dimensional array of run-time (non-constant) length. Let I be the loop control variable that ranges from 0 to the upper bound.

The location expression for VEC within the body is just the base address of the vector as usual.

The expression in the DW_AT_mapping list entry looks like

```
! Location to be mapped already pushed
Push loc (&VEC[I])              ! May require several operations
Push loc (R8)
Push 4
DW_OP_map

Push loc (&VEC[I+1])
Push loc (R9)
Push 4
DW_OP_map
```

Recall that the loop control variable I will step through the loop using an increment of 2 (the loop unrolling factor).

Nothing special is needed to handle a dynamic and unbounded size vector.

Futures

There are several opportunities for additional operations that support the same paradigm but cater to common special cases and/or idiomatic patterns of use. These may be considered as follow-on proposals.

One obvious simple set of special cases is to define variants DW_OP_map1, DW_OP_map2, DW_OP_map4, DW_OP_map8 and possibly DW_OP_map16, corresponding to an implicit size parameter of 1, 2, 4, 8 and 16, respectively.

I also think it would be simple to define special case unroll variants in support of typical unroll factors (say 2, 4, 8, 16). This would be especially true if the ABI defined some set of registers as a sequence. One might think of these as “macros” that are functionally equivalent to a sequence of mappings in the obvious way. I have not formalized the details but the approach seems promising.

PROPOSAL

[-] In Table 2.2 of Section 2.2 Attributes Names, insert the following row:

DW_AT_mappings	Supplementary location information describing parts of an object that change location at run-time.

[-] In Table 2.3 Classes of attribute value of Section 2.2 Attribute Types, insert the following row:

maplist	Specifies the location of a mapping list.

[-] In [Section 2.5], change the count of the number of DWARF storage kinds from six to five. Further, delete the last bullet at the end of the section which defines composite storage.

[-] In Section 2.5.1.7[3.17] Special Operations, add the following at the end of the section:

> Other special operations include the mapping operations which are defined only for use in mapping lists (see Section 2.6.3[3.20]).

[-] Delete Section 2.6.1.2[3.12] Composite Locations in total.

[-] Insert new Section 2.6.3[3.20] Mapping Lists

> 2.6.3[3.20] Mapping Lists
>
> Mapping lists are used to supplement the location information given in location lists. They describe the actual location of a piece of an object that has moved from its original or nominal location within an object during execution.
>
> *Location lists describe the location of a complete object while mapping lists describe the location of a piece of a complete object that has moved to another location (normally outside of the object such as to a register).*

2.6.3.1[3.20.1] Mapping List Representation

Mapping lists are contained in a separate object file section, along with location lists and value lists (see Section {locationlists}).

A mapping list is indicated by an attribute whose value is of class *maplist* (see Section {classesandforms}).

A mapping list ~~is~~ consists of a series of mapping list entries. The representation of a mapping list is the same as for a location list (see Section {locationlists}), except that

a) The expression of a counted location description must be one of the mapping list operations (see Section 2.6.3.1[3.12.1]).

   Any suitable operation may occur within expressions that push operands on the stack for use by a mapping operation.

b) There is no default mapping description that is analogous to a default location description. That is, use of the DW_LLE_default_location entry kind is not defined in a mapping list.

A mapping list entry produces a location. Certain mapping operations also convey other information as a consequence of their evaluation.

2.6.3.2[3.12.2] Mapping List Operations

There are three groups of mapping list operations which are all very similar. They differ primarily in how their operands are expressed and how many results may be produced.

2.6.3.2.1[3.12.2.1] General Form Mapping List Operations

The general form mapping operations all take the following four operands on the stack, consisting of

a) S, the source location (TOS-3)
b) M, the starting location of the mapped range (TOS-2)
c) T, the starting location of the target range (TOS-1)
d) N, the size of both ranges (TOS)

The result R is pushed on the stack, where

R = T + M-S if M <= S < M+N;

otherwise R = S.

The operators are:

1. DW_OP_map
   DW_OP_map pops four operands from the stack, consisting of
   - S, the source location (TOS-3)
   - M, the starting location of the mapped range (TOS-2)
   - T, the starting location of the target range (TOS-1)
   - N, the size of both ranges (TOS)

   The size is given in bytes and the locations must all be byte-aligned.

   The result R is pushed on the stack, where

   R = T + M-S if M <= S < M+N;

   otherwise R = S.

   The result is necessarily byte-aligned.

2. DW_OP_bit_map
   DW_OP_bit_map takes the same operands as DW_OP_map, except that the size is given in bits and the locations need not be byte-aligned. The result is calculated in the same way as for DW_OP_map.

2.6.3.2.2[3.20.2.2] Compact Form Mapping Operations

The compact form operations are equivalent to the general form operations given in the previous section except that the last three operations are given as inline constants following the operator. To make this possible, no address within the address range of a mapping entry may overlap the address range of more than one location list entry or value list entry of the containing object. This ensures that the mapped range is based on a unique location.

The compact form mapping operations are:

3. DW_OP_mapc
   DW_OP_mapc pops one operand from the stack, consisting of S, the source location. There are also three inline operands following, consisting of
   - M, the starting location of the mapped range, an unsigned LEB value relative to the nominal address of the containing object as specified by the DW_AT_location attribute of the containing object.
   - T, the starting location of the target range, an unsigned LEB128 value for a register number as defined by the architecture ABI
   - N, the size of both ranges, an unsigned LEB128 value

   The offset to the beginning of the mapped range and size are given in bytes. The size of the specified register must be equal to or larger than the size of mapped range.

   The result R is pushed on the stack, where
   R = T + M-S if M <= S < M+N;
   otherwise R = S.

4. DW_OP_bit_mapc
   DW_OP_bit_mapc takes the same operands as DW_OP_mapc, except that the offset to the beginning of the mapped range and size are given in bits. The result is calculated in the same way as for DW_OP_map.

2.6.3.2.3[3.20.2.3] Multi-Target Mapping Operations

All of the mapping operations given above provide a single result. However, immediately after the instruction that copies an in-memory component into a register, it may be that the original component remains alive and may possibly be accessed later. In rare cases, there may be more than one target location, all of which contain the same value as the component in the parent variable.

In practice, a location that is dead, that is, whose contents is never used again should not be included in the location list(s) or result(s) of a mapping list of an object even though its most recent value may persist in that location for some time.

The multi-target operators are:

5. DW_OP_maps
   DW_OP_maps is identical to DW_OP_map except that in addition to the result, the source location continues to be alive (when not the same as the result).

6. DW_OP_bit_maps
   DW_OP_bit_maps is identical to DW_OP_bit_map except that in addition to the result, the source location continues to be alive (when not the same as the result).

7. DW_OP_mapm
   DW_OP_mapm pops four + C operands from the stack, where C is the value of the second operand on the stack. The operands consist of
   - S, the source location (TOS-C-3)
   - M, the starting location of the mapped range (TOS-C-2)
   - T<C>..T<1>, the starting locations of C target ranges (TOS-1..TOS-C-1)
   - C, the unsigned count of the number of target ranges (TOS-1)
   - N, the size of all ranges (TOS)

   The size N is given in bytes and the locations must all be byte-aligned.

[ASIDE: C = 0 is a null operation and C=1 is redundant with DW_OP_map, thus is not useful. Thus, C could be biased by 2 to allow a barely more compact representation in a few cases. This does not seem worthwhile.]

The result R is pushed on the stack, where

R = T<1> + M-S if M <= S < M+N;

otherwise R = S.

In addition, for each T<C>, T<C> + M-S becomes or remains alive.

The result is necessarily byte-aligned. Unlike DW_OP_maps, the source location is treated explicitly by this operator. If the source location should be kept alive, it must be included again among the set of target locations.

8. DW_OP_bit_mapm
   DW_OP_bit_mapm takes the same operands as DW_OP_mapm, except that the size is given in bits and the locations need not be byte-aligned. The result is calculated in the same way as for DW_OP_mapm and all targets become or remain alive as for DW_OP_mapm.

A compact form of DW_OP_bit_mapm is not provided because multiple targets are rare and the inline operands would have to be in a different order (with the count of targets before the sequence of targets rather than after).

[-] In Section 4.1[5.1] Data Object Entries, insert the following after item 4:

9. A DW_AT_mappings attribute, whose value describes the pieces of an object that have moved, relative to the object as a whole, to another location at run-time.

   If no DW_AT_mappings attribute is present then the object is represented as a single consecutive sequence of bits or bytes throughout its lifetime (if present at all).

[-] In Table 7.5[8.5] Attribute Encodings of Section 7.3[8.3], add new row for

DW_AT_mappings ‡	0x??	maplist

[-] In Section 7.5.5[8.5.5] Classes and Forms, insert before the entry for rnglist:

- maplist
  A mapping list (see Section 2.6.3). This class has the same representation as class loclist.

  This class is new in DWARF Version 6.

[-] In Table 7.6[8.6] of Section 7.5.6 Form Encodings, add maplist to the Classes columns for DW_FORM_loclistx.

[-] In Table 7.9[8.9] DWARF operation encodings of Section 7.7.1[8.7.1], insert the following rows for new operations:

DW_OP_map ‡	ox??	4	location, location, location, ULEB128 constant
DW_OP_bit_map ‡	0x??	4	location location, location, ULEB128 constant
DW_OP_mapc ‡	0x??	1	location
DW_OP_bit_mapc ‡	0x??	1	location
DW_OP_maps ‡	ox??	4	location, location, location, ULEB128 constant

DW_OP_bit_maps ‡	0x??	4	location location, location, ULEB128 constant
DW_OP_mapm ‡	0x??	3+C	location, location, location,...,location, ULEB128 constant, ULEB128 constant
DW_OP_bit_mapm ‡	0x??	3+C	location, location, location,…,location, ULEB128 constant, ULEB128 constant

[-] In Table A.1 of Appendix A, add the attribute DW_AT_mappings to the right column for the TAGs DW_TAG_constant, DW_TAG_formal_parameter and DW_TAG_variable.

[-] In Appendix D Examples, add a new Section D.? Mapping List Examples.

This proposal does not contain a fully developed proposal for this new Section. However, it is hoped that, with just light editing, the Discussion section is a suitable beginning. Further, the part about accessing VEC[I] can be expanded into a supplement for the example of Issue 211206.1 Add lane support for SIMD/SIMT machines. That example (see Figure D.87 in Section D.18 SIMD Lane Example of any recent version of the DWARF V6 Working Draft) omits any description of parameters dst and src; this example would focus on dst and src.