Pointer Metadata using Multi-Mapped memory

Multi-mapped memory can be used to add metadata bits to a pointer without changing which object it points to. ZGC uses it to implement transparent support of its colored pointers. First, here’s a description of how metadata bits can be added to pointers without using multi-mapped memory or extra hardware support.

Metadata bits without multi-mapped memory

In an ordinary pointer all the bits are used to describe the address of an object. Two pointers with different bit patterns will point to different objects. Take the following to pointers as an example: 0x13210 and 0x23210. They point to two different objects.

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address bits: 0x13210</td>
<td>0001 0011 0010 0001 0000</td>
</tr>
<tr>
<td>Address bits: 0x23210</td>
<td>0010 0011 0010 0001 0000</td>
</tr>
</tbody>
</table>

When adding metadata bits to pointers we don’t want those bits to change which object it points to. So, the bits of the pointers are split into two parts, one for the address of the pointed-to object, and one for the metadata. We can choose where we make the split, but for the following examples, let’s use the 16 least significant bits for the object address, and the rest of the bits for the metadata. This gives us this pointer layout:

\[ 0xM..MAAAA : mmmm ... mmmm aaaa aaaa aaaa aaaa \]

\( (M/m = \text{metadata}, \ A/a = \text{address}) \)

Using the values from the two pointers in the example above, we get the following after the split:

<table>
<thead>
<tr>
<th>Pointer value:</th>
<th>Metadata bits:</th>
<th>Address bits:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x13210</td>
<td>0x1</td>
<td>0x3210</td>
</tr>
<tr>
<td>0x23210</td>
<td>0x2</td>
<td>0x3210</td>
</tr>
</tbody>
</table>

Here, the two pointers 0x13210 and 0x23210 refers to the same object, which is located at address 0x3210. The pointers contain the metadata 0x1 and 0x2, respectively.

We can’t directly access the contents of the object at 0x3210 through the pointers 0x13210 or 0x23210. First we need to remove the metadata bits to get the correct address, and then we can access the pointed-to memory (dereference).

```c
// Pointer with 16 address bits
ptr_with_metadata = 0x13210;

// Remove metadata bits to get the address 0x3210
AddressBitsMask = ((1 << 16) - 1);
address = ptr_with_metadata & AddressBitsMask

// Dereference and use the object at the given address
use(*address)
```

This removal of the metadata bits adds CPU instructions to the generated code, and results in a slow-down of the application.

Now, let’s see how multi-mapped memory can be used to get rid of the need to remove the metadata bits when dereferencing pointers

Metadata bits using multi-mapped memory

If we take our two example pointers and separate the address bits from the metadata bits like this:

<table>
<thead>
<tr>
<th>Pointer value:</th>
<th>Address bits:</th>
<th>Metadata bits:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x13210</td>
<td>0x3210</td>
<td>0x13210</td>
</tr>
<tr>
<td>0x10000</td>
<td>0x23210</td>
<td>0x10000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pointer value:</th>
<th>Address bits:</th>
<th>Metadata bits:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x23210</td>
<td>0x3210</td>
<td>0x23210</td>
</tr>
<tr>
<td>0x20000</td>
<td>0x3210</td>
<td>0x20000</td>
</tr>
</tbody>
</table>
Then map the same 16 bits (64 KB) of allocated memory starting at the address given by the metadata bits:

```
+-----------+ 0x10000 -----+  \\
| Mapping 1 | \       +----> +------------------------+  \\
| 0x10000   |       /  \\
+-----------+ 0x20000 -----+ 64 KB allocated memory  \\
| Mapping 2 |       +------------------------+  \\
| 0x10000   |        /  \\
+-----------+ 0x30000  \\
```

Then the address bits would act as an offset into the allocated memory. Both 0x13210 and 0x23210 would refer to the object located 0x3210 bytes into the allocated memory area.

```
+-----------+ 0x10000  \\
|          | \       +----> X @ offset 0x3210  \\
+-----------+ 0x20000  \\
| X 0x13210 | \       +------------------------+  \\
+-----------+ 0x30000  \\
```

And now we have the sought after property that independent of what metadata bits we have in the pointers we will get to the same object.

**Considerations**

**Restricts maximum addressable memory**

The examples above show two possible metadata bits values, but it is possible to use more. We need one mapping per metadata bits value that the code uses. However, there are some limitations to how many metadata bits get access to when using multi-mapped memory as described. The limitation exists because the virtual memory address range usable by user processes are limited to less than 64 bits. On Linux x64 with a four-level page table, the user process can "only" access 128TB of memory. The 47 least significant bits is what's available. The rest of the bits must be 0. This means that an application using multi-mapped memory for metadata bits, must be able to fit both the metadata bits and the object offset bits within the available 47 bits. Every extra metadata bit used halves the available offset bits and in effect the max amount of memory that can be addressed.

**Tools double-count the memory**

Tools like `ps` and `top` reports to Resident Set Size (RSS) of a process. From `man ps`:

```
rss RSS resident set size, the non-swapped physical memory that a task has used (in kilobytes) ...  
```

Even though it says that it reports physical memory, the reported value includes all virtual memory areas that map to the same physical memory.

**Requires support from OS and/or hardware**

Normally when you allocate a large chunk of memory you request anonymous memory through system calls like `mmap` or `VirtualAlloc`. You either specify the virtual address range you'd like, or you ask for a size without specifying where the memory should be allocated. You get no "handle" to the physical memory that is going to "back" the returned virtual memory range.

When memory is multi-mapped, we want better control over the physical memory and the virtual memory. We first allocate physical memory, and then map multiple virtual memory address ranges to the physical memory. This means that we need to use other APIs compared to when anonymous memory is allocated. Some of these APIs have not always had support for all features; for example large pages. Other APIs are fairly new and are not available on older OS versions.

Some CPUs support a hardware solution for masking of metadata bits. See AArch64 Tagged Address ABI and Intel Linear Address Masking (LAM). One potential drawback of using this is that it is less flexible in the number of metadata bits that can be used.

**TLB Pressure**

Accessing the same memory through multiple virtual memory address could put more pressure on the TLB. Note that ZGC puts restrictions on what metadata values a pointer may have before being dereferenced, so it's usage of multi-mapped memory isn't hitting this problem.