A Memory model with Synchronization based type aliasing V1.

N3243 For the ISO wg14 C standard.

By Eskil Steenberg Hald - eskil ‘at’ quelsolaar ‘dot’ com

This is the first draft of a proposal that could have considerable scope. It is in no way complete, and is written to convey some basic ideas and possible ways of adoption. A completed proposal will require far more discussion and consideration.

In the examples in this paper we will use the variable name "ip" for a pointer to int, and "fp" for a pointer to float.

Why Aliasing matters

In this paper we will discuss a possible way to solve a protracted problem in C's memory model.

C has two competing objectives, it wants to differentiate between different memory areas (detect Aliasing), using their type, and it wants to be able to access the same memory using different types. Let's talk about the need to separate different memory areas. Consider:

```
for(i = 0; i < *a; i++)
    b[i] = 0;
```

In this loop, we have to dereference the pointer "a" for each loop, which is slow. Ideally we would like to read the pointer only once, and transform the code in to:

```
temp = *a;
for(i = 0; i < temp; i++)
    b[i] = 0;
```

or better yet:

```
memset(b, 0, (sizeof *b) * *a);
```

However, this is not possible if a and b points to the same memory. if b[i] = 0 over writes the content of *a then the above transformations are illegal. Because of this, implementations want to be able to establish if 2 pointers are used to address the same memory area, or not. This is known as aliasing.

Essentially what the implementation would like to be able to discern is if the operation of writing to b is independent from the reading from a. Consider:

```
*a = 0;
*b = 1;
```

if a and b, are guaranteed to point at different regions of memory, then these two operations are independent and can be rearranged, but if they may point to the same memory region, then the implementation has to execute them in prescribed order.

The C standard has a few mechanisms that lets the implementation know that two pointers do not alias. The first is the keyword "restrict" that lets the user explicitly tell the compiler that a pointer. Unfortunately this is a rarely used keyword. Another way the compiler can determine that memory does not overlap is using "provenance". Essentially this means the compiler can track where the pointers come from and determine that two pointers have the same origin. Provenance is a concept that is derived from a wide spread interpretation of the standard, but where a lot of details are a bit unclear. (Jens Gustedt paper N3005 "A Provenance-aware Memory Object Model for C" does a good job of clarifying provenance)

Provenance is of specific interest for this paper, because it tries to codify something that implementations already do, but that the standard is less clear about.

Effective type is conceptually broken.

C has a rule that says that once memory has been written to, it is assigned an "effective type" and the memory can then only be read back using the same type. This rule is fundamentally broken for a number of reasons, and it should be removed and replaced with something different.

-This entirely goes against user expectations. Most programmers expect to be able to write to memory using one type and read with another. Vast amounts of code relies on being able to do this.

-The concept of "Effective type" is fundamentally counter to how computer architectures work. Memory stores data, not the type of the data that is stored (computers may have different registers for different types, but not main memory). The type is determined by the operations we do on memory. One could imagine a architecture that uses different types of memory for different types of types, but C would not be implementable on such a platform because C has generic allocators (malloc, realloc, calloc) that don't require the declaration of type, interleaved types (structs), and effective type concept does let the user reuse memory for a different type, so the effective type rule would prevent such a platform anyway.

-The model at its core does not take into account that memory can be copied, read and written using standard library functions like fread, fwrite and memcpy. A number of complex exceptions have therefore been made, to make these functions work. Because these functions have been given special exceptions, they can not be implemented in ISO C. Example:

```
memcpy(a, b, sizeof(float));
```

does not have the same semantic meaning as:

```
memcpy(a, b, 4);
```

even on platforms where "sizeof(float)" evaluates to 4. (the memcpy with a sizeof(float) actually assigns an effective type to the memory, according to the standard) There is also a somewhat arbitrary exception for unsigned char, so that functions that read memory like fread can work.

-The purpose of the effective type rules are to aid in alias analysis, but even here they fall apart. consider, the a pair of int and float pointers:

```
*ip = 0;
*fp = 1;
if(x)
   *ip = 2;
```

If the rules stated that pointers of different types can not point to the same memory, ip and fp, would be entirely independent, and the compiler could rearrange these then independently of each other. However since the rules only state that as long as the read operation has the same type as the written type it can't rearrange these operations because they are not independent.

-Ambiguities about the rules remain about structs and unions. If memory is written to as an int that is a member of a struct, can it be accessed using a pointer to an int, or does it have to be accessed using a compatible struct?

-Very few people know about this rule, and therefore an enormous amount of existing code is technically UB.

-Because of all these issues, no implementation currently implements the effective rules. Doing so would break too much existing software.

-Several people in the wg14 have expressed their dislike of "effective type" and the need for an alternative solution.

Our argument is to scrap the effective type concept in its entirety.

Type pruning

Consider the following code:

```
*fp = 3.14;
ip = fp;
i = *ip;
```

This code breaks the effective type rules and is UB according to C99 and later. However, in reality all known compilers handle this the way users expect it to. So where is the issue? Why can't we just make this code standard compliant? The issue is that in this simple case it is trivial to see that ip and fp alias. In more complex code the compiler may have to track the aliasing from far away. There are ways for the user to write code that makes this almost impossible. Proving that they do alias is easy, but what compilers need to be able to do is to prove that they do not in order to optimize and that is far more difficult.

The ability to read memory as a different type from what it was written is fundamental to a lot of uses of C. It prevalent in existing code, and most C programers have no idea that the standard puts restrictions on this. The only way to do type pruning legally between arbitrary types is to use a union. In practice this isnt always feasible, because an interface may be able to in advance know what types will be given to it, and because it in many cases requires the program to copy the data into a union, in order to perform the prune, and then copy it again to its destination.

Because the practice of casting pointers to perform type pruning is so prevalent, all compilers give users leeway to do so to some degree. However how much leeway is given is not easily known. To guarantee that the program does not step over the line options like -nostrictalaising have been added and adopted by users.

Proposed solution

Our proposal is to add a rule that says that anytime a pointer is being converted from or to, we create an aliasing barrier between the two types. At this singular point, the two types alias, and thorfor operations can not freely pass this barrier. In concept this is very similar to the atomic properties. Example:

```
*fp = 3.14;
ip = fp; // barrier
i = *ip;
```

By giving the pointer conversion this new aliasing barrier property, we guarantee that any operation on memory that may alias with fp, has to be completed before the conversion, and that any operation that may alias with ip, cant happen until after the conversion. You could say that converting from a pointer type has a "alias release" semantic, and a pointer being converted to has a "alias acquire" semantic.

Because the synchronization happens at only one specific point, the following will be UB:

```
ip = fp;
*fp = 1;
*ip = 2;
```

because the compiler is free to rearrange the last two assignments since fp and ip no longer alias. This means that in some cases on may need to add additional assignments of pointers to convey an aliasing barrier. Consider:

```
*fp = 3.14;
ip = fp;
++(*ip);
fp = ip;
printf("%f\n", *fp);
```

In this case we synchronize ip and fp in order to do an integer operation on a floating point representation, but then we need to synchronize it back in order to read out the floating point representation.

Since function arguments in the form of pointers can be used both as input and output, andy conversion to a function argument would have to be a full barrier (acquire and release).

This would enable all kinds of user space memory management, user provided implementations of memcpy and memset that are conformant and use larger types then unsigned char to move memory, it would enable encryption of arbitrary data using large types in place and much more. All things most C programmers think are already permitted by the standard.

Implications

This solution would remove a lot of the complexities of the memory model. It would turn a lot of code that is today UB well defined, but some code would remain UB. However, there would be a clear way to make that code conformant, that would only require the movement or addition of pointer conversions.

However: This change WOULD BREAK CONFORMANT CODE. Yes, strictly speaking this would be a change that is NOT backwards compatible. Consider:

```
ip = fp;
*fp = 1;
```

```
*ip = 2;
```

As long as the content of the pointer is read using the same type as it was written (int) this code is conformant using the effective type rules. The compiler can not rearrange the last two statements. However, if we remove the effective type rules, and instead replace them with our proposal, the code becomes UB, because the implementation is free to rearrange the last two statements.

Implementation of change

In reality, our expectation is that this change would not result in breaking existing code, because implementers would simply not try to optimize for it. Even if the C memory model becomes 100% clear, implementations will want to continue to compile existing code that is technically UB.

Because compilers do not implement effective type, and try to track pointer conversions, we would expect existing implementations to mostly already be conformant to this change. Because users are required to do the conversion between the usage of the differently typed pointers, the type conversion cant be far away. Obviously an implementation would have to verify this.

The change to existing implementations should be minimal if this change is adopted. We expect implementers to simultaneously support: the new model, the old model, and existing code. While the new model adds some requirements for implementers, these requirements are very minor, compared to the requirements to support non-conformant existing code.

However for users, there now exists clear guidelines for how to do what they want to do while being guaranteed to be conformant. This is far better than relying on: "Sure it technically UB, but no one in their right mind would ever write a compiler that does not accept this, so its ok".

Alternative memory model

No matter how one may choose to frame the impact of this proposal, the reality is that it breaks existing code, even if implementations don't do it. While it is a great improvement for users to have clear guidelines for how to be standard compliant, it doesn't help implementers if this is just a subset of what they have to support.

If compilers can't trust that code is written within the rules of the standard, their job becomes far more difficult, and they have to leave a lot of optimization opportunities alone, for fear of breaking existing code. Currently compilers cant follow the standard because it would simply be unacceptable to their users. It would break too much code, and reimplementing existing code to conform to effective type, and other rules, would in many cases require significant re architecture, to be feasible.

The C memory model, have many other issues like:
-When can a volatile write be torn?
-C does not respect any concurrent with external libraries like pthreads.
-Numerous issues with the concurrency model, like dependent reads.
-Adoption and specification of provenance rules.

Our current thinking is that, perhaps the best way forward is to define an alternative memory model for C, based on these ideas, current implementations, existing code, and borrowing some from the Linux kernel memory model, and using external standards such as posix for concurrency. The goal would be to formalize a memory model that is more formal and as close to existing implementations and practice as possible. Existing implementations would be very close to already conformant to this model, in some more conformist than to the existing model.

This would enable implementations to continue to support existing code bases, and be 100% compliant with the old and alternative memory model, and also provide options like -strictalternativemodel that would let them freely optimize, while giving users the ability to do the thing they need to do. This would serve both users and implementers by giving the language a sound footing. Additionally an alternative memory model could be applied to any version of C not just the latest one. Given that many users and open source projects use previous versions of C this would be desirable.