

# MEM-REF and beyond—desired changes to the GIMPLE IL

Richard Günther  
*Novell, SUSE Labs*  
rguenther@suse.de

## Abstract

We present the motivation behind the recent GIMPLE IL change introducing MEM-REF and the challenges in enforcing its use. In particular rehashing type-based alias analysis and the memory model used by the GCC middle-end proves interesting in this context.

After presenting an overview of the past evolution of the GIMPLE IL we present multiple desired changes to it, mostly motivated by the link-time optimization framework which faces the problem to deal with multiple source languages, and the desire to move work from RTL expansion and the RTL pass pipeline up to the GIMPLE level.

## 1 Introduction

The GIMPLE intermediate language has undergone several representation and semantics changes since its introduction in 2004[2]. The most visible one was the move from representing GIMPLE statements using the ubiquitous tree structure to the tuples representation.

Less visible was the move towards fully defined semantics for all parts of GIMPLE, making all frontend specifics explicit. This includes introduction and continuous adjustment of a middle-end type-system, the introduction of a middle-end memory-model and the attempt to define type-based aliasing rules independent of language semantics.

This eventually allowed the link-time optimization framework LTO to work in less of an undefined territory when optimizing mixed-language programs. With that in mind the compilation process now tries to separate frontend and middle-end processing strictly with the process of gimplification, the translation of the GENERIC intermediate language to the GIMPLE intermediate language.

In the following we will outline the motivation and implementation of the latest semantic and structural change to the GIMPLE IL, the introduction of the MEM-REF operation. After that further generally desired changes to GIMPLE are proposed.

## 2 MEM-REF

With GCC 4.6 a new tree code, `MEM_REF`, is introduced which is supposed to be used as base for all memory accesses in future releases.

Introducing a MEM-REF like operation has been tried before in a more radical manner. With MEM-REF as present in GCC 4.6 we take a more incremental approach of eventually arriving at the same destination. The initial MEM-REF implementation tries to solve several problems which we will lay out in detail now.

Before MEM-REF memory operations could have several flavors of INDIRECT-REFs as bases all of which require type-correct pointer operands for TBAA purposes. With MEM-REF the use of the pointer operand as value and that as carrier for TBAA information is separated, allowing value-preserving conversions of pointers to be omitted from GIMPLE.

For alias-analysis purposes it is desired that as much information about a memory access is directly available in the memory access tree. This includes information what a pointer used in an indirect memory reference points to. Information about the pointed-to object can be obtained by traversing the SSA use-def chain of the pointer. It often happens that abstraction in the source program results in indirect memory accesses to declared objects. If that is visible via the SSA use-def chain combining the address-taking operation with the indirection will result in an easier to analyze intermediate language.

Before MEM-REF re-combining memory accesses to make the declared object visible required to be type-correct, thus undefined type-punning was forced to be

less apparent which caused us to optimize code in unexpected ways more often than necessary. Re-combining memory accesses pre-MEM-REF also changed TBAA information in subtle ways, losing the original effective type of the access. For example when combining `p=&q->x` with `*p` to form `q->x` GCC would treat `*q` as the base to extract TBAA information from even though `&q->x` might merely act as offsetting operation.

MEM-REF now combines pointer indirection and direct object access within the same abstraction and adds a constant offset operand to allow propagation of the address of declared objects in more cases.

## 2.1 MEM-REF operation

The MEM-REF operation has two operands. The first operand is a pointer which specifies the access location base. The second operand is a constant byte offset which is added to the access location base. The type of the constant is a pointer type which, when dereferenced, specifies the effective type of the access and thus its alias-set. The type of the MEM-REF operation specifies the type of the memory access, its size and alignment.

```
vector(4) int vect_inter_high.154;
vector(4) int vect_var_.17;
vect_var_.17_76
  = MEM[(int[900] *)D.2339_58 + -512B];
MEM[(int[64][16] *)D.2354_105]
  = vect_inter_high.154_230;
```

Here are two examples of MEM-REF operations as visible in dump files. The first one loads from the location `D.2339_58-512` with the effective access-type `int[900]`, producing an integer vector value. The second statement stores to the location `D.2354_105` with the effective access-type `int[64][16]`, a memory location of integer vector type.

## 2.2 MEM-REF Implementation

The MEM-REF operations are introduced during simplification. At that point TBAA information from memory accesses using indirect references are extracted and the `INDIRECT_REF` base is rewritten using a MEM-REF operation using the original pointer and constant offset zero of the extracted TBAA relevant type.

After simplification all memory accesses have to be either based on MEM-REF (or TARGET-MEM-REF) or on plain declarations.

## 2.3 Limitations and future work

Using a MEM-REF as base for a memory access is not yet enforced, directly using a declaration is still permitted. Similarly a MEM-REF may still be subsetted via component references even if that only adds a constant offset. Eventually both will be disallowed in future releases.

MEM-REF can not yet directly encode non-constant offsets and thus cannot be used as a full replacement for all reference class trees without forcing pointer indirection and introducing additional statements to compute the pointer value. As TARGET-MEM-REF has been adjusted to be similar to MEM-REF that can be used for this purpose to some extent.

MEM-REF cannot be used to encode bit-field accesses. At the moment subsetting with a component reference or a bit-field reference is required. Eventually bit-field accesses will be lowered to byte-aligned loads plus bit-field extraction on registers and byte-aligned read-modify-write operations.

Similar to the MEM-REF changes changes to call statements are necessary to preserve the original type of the called function as the frontend specified it. This would avoid function pointer conversions to stay in the IL. Conveniently this type could be encoded in the call statement tuple instead of some tree operand.

## 3 Desired Changes to the GIMPLE IL

In this section we outline several areas in the GIMPLE intermediate language which are problematic. Possible solutions are presented and motivated. Most of the issues are well-known and some have been even worked on in the past.

### 3.1 TYPE\_IS\_SIZETYPE

There is a class of integer types in GCC that are special. These are the so-called *sizetypes*, types which have the `TYPE_IS_SIZETYPE` flag set. These types have special semantics that are not documented or even agreed upon. The only documented specialty is that all size-type constants are sign-extended regardless of the sign of their type. Sizetypes also have special behavior on integer overflow in that it is both undefined but also never happens.

The main question is why would extension behavior matter, or rather, in which case would a `sizetype` value be extended? Obviously we now run into this issue with the forced use of `sizetype` for the `POINTER_PLUS_EXPR` offset operand. For targets where the size of `sizetype` does not match that of pointers such extension happens during RTL expansion.

Other uses of `sizetype` come from uses in sizes and offsets of types and declarations.

The proposed change is to end all the weirdness and drop `TYPE_IS_SIZETYPE`.

### 3.2 `POINTER_PLUS_EXPR`

Currently `POINTER_PLUS_EXPR` requires the offset operand to be of `sizetype`. This is problematic because `sizetype` is of the special `TYPE_IS_SIZETYPE` kind, whose uses should go away. Second, there are targets where `sizetype` is of different precision than pointer types which requires sign- or zero-extension of the offset operand depending on its sign which we just lost by forcing it to be of `sizetype` type.

There are two possible ways out. First, introduce a new type, `ptrdifftype`, which is always of the same size as pointers and thus does not require extension.

Second, change `POINTER_PLUS_EXPR` to accept any kind of integer type for the offset operand which would be extended properly at expansion time. `POINTER_PLUS_EXPR` would thus have an embedded implicit extension to something like `ptrdifftype`.

### 3.3 `COND_EXPR`

Currently `COND_EXPR` is created by if-conversion. `COND_EXPR` is one of the tree codes that are handled as `GIMPLE_SINGLE_RHS`.

The proposed change is to make `COND_EXPR` `GIMPLE_TERNARY_RHS` and to split out predicate computation to a separate statement.

### 3.4 Separating predicate computation

Currently predicates are computed within `GIMPLE_COND` statements, `COND_EXPR` rhs of `GIMPLE_ASSIGN` and as separate computations in `GIMPLE_`

`ASSIGN`. The proposed change is to move those from `GIMPLE_COND` and `COND_EXPR` into separate `GIMPLE_ASSIGN` statements making the predicate result available as an SSA name.

The immediate benefit is consolidate code that looks at predicates and tries to simplify them. Another benefit is that the predicate values get assigned to SSA names and thus are exposed to SSA based optimization passes such as (partial) redundancy elimination which might in future drive jump threading.

### 3.5 Enumeration type min- and max-values have semantics

Enumeral types are the only type kinds where the types min- and max-value when it disagrees with the types precision have a semantic. Value-range propagation uses it to derive value-ranges. That semantic isn't taken into account by the gimple type-system which treats the different integer type kinds as equal and only distinguishes based on type precision. The proposed change is to drop enumerals min- and max-value from having a semantic meaning in the middle-end.

### 3.6 Drop undefined signed overflow

Signed overflow is treated as undefined by the middle-end. This complicates and penalizes optimizations in that optimizers need to be careful to not introduce signed arithmetic that might overflow. Most optimization passes thus disable themselves for signed arithmetic or produce unsigned arithmetic and thus casts to and from unsigned variants.

The proposed change is to make all integer and pointer arithmetic in the middle-end wrapping. To preserve and improve optimization opportunities when range information is available for the operands of arithmetic operations this information should be encoded explicitly in the intermediate language. On the no-undefined-overflow branch operation code variants that state that the operation does not wrap are introduced. This is enough to communicate signed overflow undefinedness from the frontend to the middle-end as well as to keep some value-range information derived by propagation persistent.

### 3.7 Aggregate uses should be separated

In GIMPLE generally loads and stores are performed as separate statements if they involve types that are suitable for register promotion. As GIMPLE does not allow aggregates to be in SSA form separating the load from the store in a copy operation of an aggregate type is not possible (without recursively introducing another aggregate copy).

Currently aggregate uses can appear in `GIMPLE_CALL`, `GIMPLE_ASM` and `GIMPLE_ASSIGN`. This complicates code that needs to handle loads and stores as well as making store and load associated data such as alignment difficult to place at the statement level.

The most elegant solution, first prototyped by the middle-end array work[1], is to introduce temporary aggregates in SSA form to split the copy operations. This allows simplifying passes and some scalar optimizations be performed on aggregates.

More specifically,

```
struct X x, y;
# .MEM_2 = VDEF <.MEM_1>
x = y;
```

would be lowered to

```
struct X x, y, tem;
# VUSE <.MEM_1>
tem_3 = y;
# .MEM_2 = VDEF <.MEM_1>
x = tem_3;
```

similarly

```
struct X x, y;
# .MEM_2 = VDEF <.MEM_1>
x = foo (y);
```

would become

```
struct X x, y, tem;
# VUSE <.MEM_1>
tem_3 = y;
# .MEM_2 = VDEF <.MEM_1>
tem_4 = foo (tem_3);
# .MEM_5 = VDEF <.MEM_2>
x = tem_4;
```

### 3.8 Aggregates should be rewritten into SSA form

Aggregates that are never partially assigned to can be re-written into SSA form. That allows scalar optimizers

such as copy-propagation to work on them and possibly reduce the burden on memory optimizers and stack usage.

Eventually this would introduce reference-class trees being used with SSA name bases on the right-hand side of assignments which might be unwanted. Thus restricting this to cases where no partial access at all appears might be an easy intermediate step.

Implementation-wise the `DECL_GIMPLE_REG_P` flag can be used to mark register variables. This flag could be made mandatory also for register-typed variables when their address is not taken and in turn would simplify the implementation of the `is_gimple_reg` predicate.

### 3.9 Aggregate lifetime lowering

When lowering scope blocks and bringing all local variables to function scope the places of aggregate destruction should be preserved by placing a note in the IL. This allows the stack-slot sharing machinery to more precisely compute the lifetime of aggregates and drop relying on scope-block information that gets out-of-date with code-motion optimizations.

The easiest way to annotate the point an aggregate dies is to assign to it from a special undefined value. As a simple example a constructor with a single error-mark element could be used.

```
{
  struct X x;
  ...
  # .MEM_2 = VDEF <.MEM_1>
  x = { <error_mark_node> };
}
```

### 3.10 Return statements should have a VUSE

Returning from a function keeps stores to non-local memory life but this is not represented in the virtual FUD chain.

Assigning a `VUSE` to the return statement of a function makes it possible to walk all stores that reach the function exit. Likewise a `VUSE` at the return statement ensures that PHI nodes for virtual operands are inserted towards function exit which is required to perform store sinking properly as in the example below.

```
# .MEM_2 = VDEF <.MEM_1>
a = 1;
if (p)
  # .MEM_3 = VDEF <.MEM_2>
  a = 2;
return;
```

The lack of a PHI node merging `.MEM_2` and `.MEM_3` makes the store sinking implementation needlessly complicated.

## References

- [1] Richard Guenther. Middle-end array expressions. In *GCC Developers' Summit*, June 2008.
- [2] Diego Novillo. Design and implementation of tree ssa. In *GCC Developers' Summit*, June 2004.