MEM-REF and beyond – desired changes to the GIMPLE IL

Richard Günther

Oct. 25th, 2010

Contents

GCC Intermediate Language History

pre tree-SSA

- Expand converts GENERIC (fronted) to RTL (middle-end)
- Optimizers work mostly on RTL

tree-SSA

- Gimplifier converts GENERIC (fronted) to GIMPLE (middle-end)
- Expand converts GIMPLE (middle-end) to RTL (middle-end)
- Most optimizers work on GIMPLE

GCC Intermediate Language History

pre tree-SSA

- Expand converts GENERIC (fronted) to RTL (middle-end)
- Optimizers work mostly on RTL

tree-SSA

- Gimplifier converts GENERIC (fronted) to GIMPLE (middle-end)
- Expand converts GIMPLE (middle-end) to RTL (middle-end)
- Most optimizers work on GIMPLE

GIMPLE?

The long way of lowering the IL...

- Start with High GIMPLE
- Lower nested functions
- Lower OpenMP
- Lower lexical scopes, produce Low GIMPLE
- Lower EH constructs
- Build a CFG
- Lower generic vector operations
- Build (memory) SSA form

Which is what optimizers work on.

GIMPLE?

The long way of lowering the IL...

- Start with High GIMPLE
- Lower nested functions
- Lower OpenMP
- Lower lexical scopes, produce Low GIMPLE
- Lower EH constructs
- Build a CFG
- Lower generic vector operations
- Build (memory) SSA form

Which is what optimizers work on.

GIMPLE?

The long way of lowering the IL...

- Start with High GIMPLE
- Lower nested functions
- Lower OpenMP
- Lower lexical scopes, produce Low GIMPLE
- Lower EH constructs
- Build a CFG
- Lower generic vector operations
- Build (memory) SSA form

Which is what optimizers work on.

GIMPLE?

The long way of lowering the IL...

- Start with High GIMPLE
- Lower nested functions
- Lower OpenMP
- Lower lexical scopes, produce Low GIMPLE
- Lower EH constructs
- Build a CFG
- Lower generic vector operations
- Build (memory) SSA form

Which is what optimizers work on.

GIMPLE?

The long way of lowering the IL...

- Start with High GIMPLE
- Lower nested functions
- Lower OpenMP
- Lower lexical scopes, produce Low GIMPLE
- Lower EH constructs
- Build a CFG
- Lower generic vector operations
- Build (memory) SSA form

Which is what optimizers work on.

GIMPLE?

The long way of lowering the IL...

- Start with High GIMPLE
- Lower nested functions
- Lower OpenMP
- Lower lexical scopes, produce Low GIMPLE
- Lower EH constructs
- Build a CFG
- Lower generic vector operations
- Build (memory) SSA form

Which is what optimizers work on.

GIMPLE?

The long way of lowering the IL...

- Start with High GIMPLE
- Lower nested functions
- Lower OpenMP
- Lower lexical scopes, produce Low GIMPLE
- Lower EH constructs
- Build a CFG
- Lower generic vector operations
- Build (memory) SSA form

Which is what optimizers work on.

GIMPLE?

The long way of lowering the IL...

- Start with High GIMPLE
- Lower nested functions
- Lower OpenMP
- Lower lexical scopes, produce Low GIMPLE
- Lower EH constructs
- Build a CFG
- Lower generic vector operations
- Build (memory) SSA form

Which is what optimizers work on.

GIMPLE?

The long way of lowering the IL...

- Start with High GIMPLE
- Lower nested functions
- Lower OpenMP
- Lower lexical scopes, produce Low GIMPLE
- Lower EH constructs
- Build a CFG
- Lower generic vector operations
- Build (memory) SSA form

Which is what optimizers work on.

Evolution of GIMPLE

- Originally same representation as GENERIC
- Optimizations on the tree representation
- New statement representation, `gimple` (Tuples)

Trees are still used for statement operands. In some places they are arbitrarily nested and have an arbitrary number of sub-operands.

Evolution of GIMPLE

- Originally same representation as GENERIC
- Optimizations on the tree representation
- New statement representation, `gimple` (Tuples)

Trees are still used for statement operands. In some places they are arbitrarily nested and have an arbitrary number of sub-operands.

Evolution of GIMPLE

- Originally same representation as GENERIC
- Optimizations on the tree representation
- New statement representation, `gimple` (Tuples)

Trees are still used for statement operands. In some places they are arbitrarily nested and have an arbitrary number of sub-operands.

Evolution of GIMPLE

- Originally same representation as GENERIC
- Optimizations on the tree representation
- New statement representation, `gimple` (Tuples)

Trees are still used for statement operands. In some places they are arbitrarily nested and have an arbitrary number of sub-operands.

Contents

Motivations

- Smaller, non-recursive memory access trees
- TBAA unlimited address combining
- Variant accesses for plain decls
- Easy non-aliased memory access construction with base and offset

Motivations

- Smaller, non-recursive memory access trees
- TBAA unlimited address combining
- Variant accesses for plain decls
- Easy non-aliased memory access construction with base and offset

Motivations

- Smaller, non-recursive memory access trees
- TBAA unlimited address combining
- Variant accesses for plain decls
- Easy non-aliased memory access construction with base and offset

Motivations

- Smaller, non-recursive memory access trees
- TBAA unlimited address combining
- Variant accesses for plain decls
- Easy non-aliased memory access construction with base and offset

Schematics

`MEM_REF`<type>(base, offset)

- type specifies size and alignment of the access and the value type
- base specifies the base address
- offset specifies a constant offset to the base address
- typeof(offset) specifies the type for TBAA purposes

MEM-REF

Summary for the casual observer

- All `INDIRECT_REF` flavors are gone
- All memory access bases can be based on a `MEM_REF`
- `TARGET_MEM_REF` is an extension of `MEM_REF`

MEM-REF

Summary for the casual observer

- All `INDIRECT_REF` flavors are gone
- All memory access bases can be based on a `MEM_REF`
- `TARGET_MEM_REF` is an extension of `MEM_REF`

MEM-REF

Summary for the casual observer

- All `INDIRECT_REF` flavors are gone
- All memory access bases can be based on a `MEM_REF`
- `TARGET_MEM_REF` is an extension of `MEM_REF`

Limitations

Limitations compared to old MEM-REF work

- Alignment is not explicit
- Size is not explicit
- Non-constant indices are not handled (`TARGET_MEM_REF`)

Limitations

Limitations compared to old MEM-REF work

- Alignment is not explicit
- Size is not explicit
- Non-constant indices are not handled (`TARGET_MEM_REF`)

Limitations

Limitations compared to old MEM-REF work

- Alignment is not explicit
- Size is not explicit
- Non-constant indices are not handled (`TARGET_MEM_REF`)

Beyond MEM-REF

During the next stage1

- Force use of `MEM_REF` as base
- Lower all invariant addresses to `POINTER_PLUS_EXPR`
- Make frontends use `MEM_REF` instead of `TYPE_CANONICAL` and `lang_hooks.get_alias_set` for TBAA

Beyond MEM-REF

During the next stage1

- Force use of `MEM_REF` as base
- Lower all invariant addresses to `POINTER_PLUS_EXPR`
- Make frontends use `MEM_REF` instead of `TYPE_CANONICAL` and `lang_hooks.get_alias_set` for TBAA

Beyond MEM-REF

During the next stage1

- Force use of `MEM_REF` as base
- Lower all invariant addresses to `POINTER_PLUS_EXPR`
- Make frontends use `MEM_REF` instead of `TYPE_CANONICAL` and `lang_hooks.get_alias_set` for TBAA

Questions

Questions?

Contents

Semantic Issues

- `TYPE_IS_SIZETYPE`
- Undefined signed overflow
- `TYPE_MIN/MAX_VALUE` and `TYPE_PRECISION`
- `POINTER_PLUS_EXPR` and its use of `sizetype`
- `GIMPLE_CALL` relies on the function pointer type

Semantic Issues

- `TYPE_IS_SIZETYPE`
- Undefined signed overflow
- `TYPE_MIN/MAX_VALUE` and `TYPE_PRECISION`
- `POINTER_PLUS_EXPR` and its use of `sizetype`
- `GIMPLE_CALL` relies on the function pointer type

Semantic Issues

- `TYPE_IS_SIZETYPE`
- Undefined signed overflow
- `TYPE_MIN/MAX_VALUE` and `TYPE_PRECISION`
- `POINTER_PLUS_EXPR` and its use of `sizetype`
- `GIMPLE_CALL` relies on the function pointer type

Semantic Issues

- `TYPE_IS_SIZETYPE`
- Undefined signed overflow
- `TYPE_MIN/MAX_VALUE` and `TYPE_PRECISION`
- `POINTER_PLUS_EXPR` and its use of `sizetype`
- `GIMPLE_CALL` relies on the function pointer type

Semantic Issues

- `TYPE_IS_SIZETYPE`
- Undefined signed overflow
- `TYPE_MIN/MAX_VALUE` and `TYPE_PRECISION`
- `POINTER_PLUS_EXPR` and its use of `sizetype`
- `GIMPLE_CALL` relies on the function pointer type

Representational Issues

- `COND_EXPR`, `VEC_COND_EXPR`, `DOT_PROD_EXPR` should be ternary RHS
- Separate predicate computations from its uses (`COND_EXPR`, `GIMPLE_COND`)
- Separate all loads and stores (including aggregates and copies)
- Allow aggregates in SSA form
- Lower bitfield accesses
- Lower array accesses

Representational Issues

- `COND_EXPR`, `VEC_COND_EXPR`, `DOT_PROD_EXPR` should be ternary RHS
- Separate predicate computations from its uses (`COND_EXPR`, `GIMPLE_COND`)
- Separate all loads and stores (including aggregates and copies)
- Allow aggregates in SSA form
- Lower bitfield accesses
- Lower array accesses

Representational Issues

- `COND_EXPR`, `VEC_COND_EXPR`, `DOT_PROD_EXPR` should be ternary RHS
- Separate predicate computations from its uses (`COND_EXPR`, `GIMPLE_COND`)
- Separate all loads and stores (including aggregates and copies)
- Allow aggregates in SSA form
- Lower bitfield accesses
- Lower array accesses

Representational Issues

- `COND_EXPR`, `VEC_COND_EXPR`, `DOT_PROD_EXPR` should be ternary RHS
- Separate predicate computations from its uses (`COND_EXPR`, `GIMPLE_COND`)
- Separate all loads and stores (including aggregates and copies)
- Allow aggregates in SSA form
- Lower bitfield accesses
- Lower array accesses

Representational Issues

- `COND_EXPR`, `VEC_COND_EXPR`, `DOT_PROD_EXPR` should be ternary RHS
- Separate predicate computations from its uses (`COND_EXPR`, `GIMPLE_COND`)
- Separate all loads and stores (including aggregates and copies)
- Allow aggregates in SSA form
- Lower bitfield accesses
- Lower array accesses

Additional IL Features

- Return statements should have a `VUSE`
- Aggregates end of life should be represented explicitly

Additional IL Features

- Return statements should have a `VUSE`
- Aggregates end of life should be represented explicitly

Questions

.

Questions?

$\longrightarrow$ BOF.