Towards a representation of arbitrary alias graph in LLVM for Fortran

—

Tarique Islam (tislam@ca.ibm.com)
Kelvin Li (kli@ca.ibm.com)

IBM

Content

- Background and motivation
- Formalization – IR representation of alias graph
- Alias.scope-noalias based representation
- Results and discussions
- Future work

Background and motivation

- IBM compiler team has been actively participating and collaborating in the LLVM project
  - Loop Opt WG, OpenMP and others
- Utilize the clang infrastructure in the products (i.e. XL C/C++ compiler on Linux)
- LLVM project extends to include Fortran language
  - flang is officially part of the LLVM repo
- Fortran language is important to the HPC community
  - performance is the key

Background and motivation

- To achieve high performance, providing precise alias information is one of the key factors
- Fortran language provides alias rules
  - disallow the same variable as two actual arguments to a procedure in which one of the corresponding arguments is accessed or modified under certain condition
- Some Fortran objects are internally represented by descriptors (or dope vectors)
  - accessing the descriptor's fields may require more precise alias information
- Alias sets may overlap
  - existing alias infrastructure may not be able to represent it
- Precise alias information helps optimization (e.g. loop optimization)

Background and motivation

- XL Fortran frontend collects and computes alias information in internal representation
- Translate the internal alias representation to LLVM IR
  - preserve the accuracy of the alias information generated by the frontend

Formalization

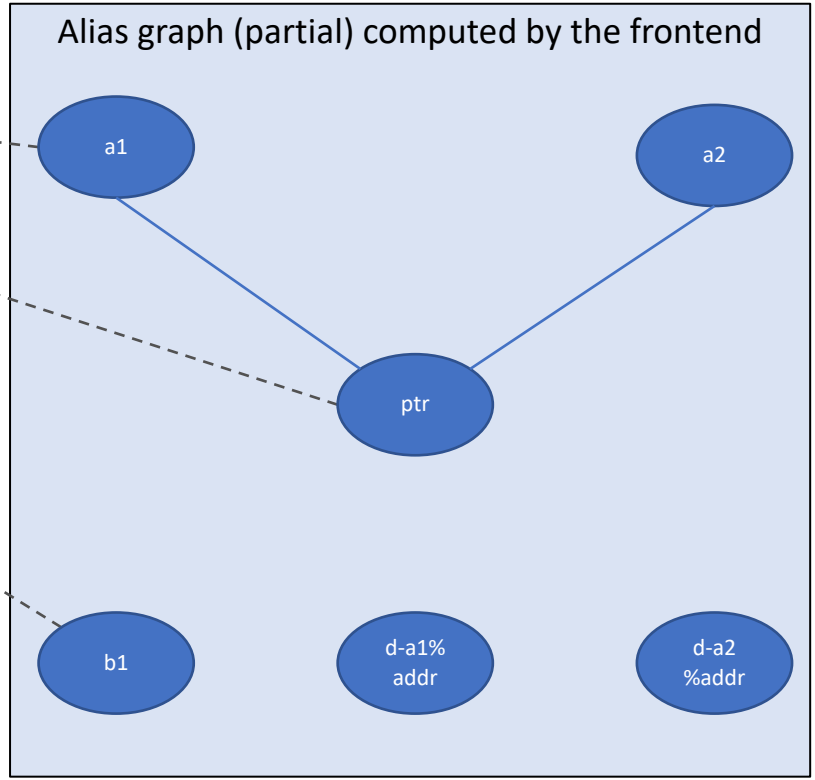
- The compiler-frontend can apply language rules to determine whether a pair of symbols in the program may alias each other
- The *may-alias* relation can be viewed as a graph where
  - each graph node corresponds to a program symbol, and
  - each graph edge represents a *may-alias* relation between the corresponding program symbols
- We refer to the above graph as the *alias-graph* of the program

Fortran example: Alias graph

```
module m
integer, allocatable, target :: a1, a2
integer, pointer :: ptr
integer :: b1
contains
subroutine init_ptr()
  ptr = 0
end subroutine init_ptr
subroutine compute(n)
  integer :: idx, n
  ...
  do idx = 1, n
    a1 = a1 * b1 + a2 * b1
  end do
end subroutine compute
end module m
```

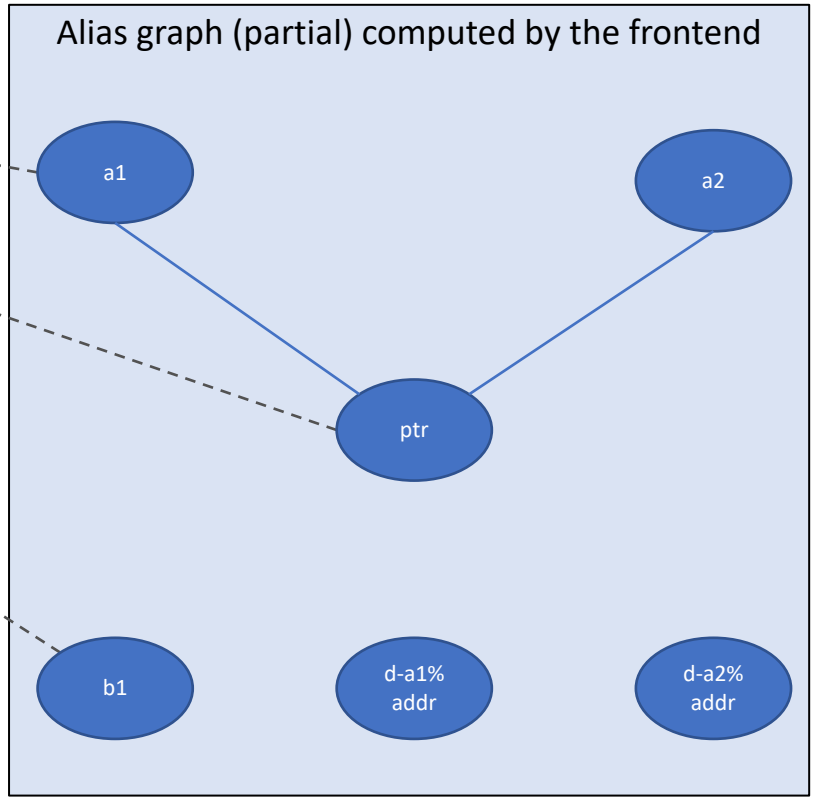
Fortran example: Alias graph

```
module m
integer, allocatable, target :: a1, a2
integer, pointer :: ptr
integer :: b1
contains
subroutine init_ptr()
  ptr = 0
end subroutine init_ptr
subroutine compute(n)
  integer :: idx, n
  …
  do idx = 1, n
    a1 = a1 * b1 + a2 * b1
  end do
end subroutine compute
end module m
```



Fortran example: Alias graph

```
module m
integer, allocatable, target :: a1, a2
integer, pointer :: ptr
integer :: b1
contains
subroutine init_ptr()
  ptr = 0
end subroutine init_ptr
subroutine compute(n)
  integer :: idx, n
  ...
  do idx = 1, n
    a1 = a1 * b1 + a2 * b1
  end do
end subroutine compute
end module m
```



IR after opt without alias information from frontend

```
Option: -O3 --unroll-max-count=0

define dso_local void @__m_NMOD_compute(i32* noalias nocapture readonly dereferenceable(4) %.n) local_unnamed_addr #1 {
__m_NMOD_compute_entry:
  tail call void @init_var(i32* getelementptr inbounds (%"_type_of_&&N&&m", %"_type_of_&&N&&m"* @"&&N&&m", i64 0, i32 0)) #4
  %"_val_d-a1%addr_" = load i32*, i32** getelementptr inbounds (%"_type_of_&&N&m", %"_type_of_&&N&m"* @"&&N&m", i64 0, i32 0), align 16
  %"_val_d-a2%addr_" = load i32*, i32** getelementptr inbounds (%"_type_of_&&N&m", %"_type_of_&&N&m"* @"&&N&m", i64 0, i32 6), align 16
  tail call void @init_alloc_var(i32* %"_val_d-a1%addr_", i32* %"_val_d-a2%addr_") #4
  %_val_n_ = load i32, i32* %.n, align 4
  %_grt_tmp2 = icmp slt i32 %_val_n_, 1
  br i1 %_grt_tmp2, label %_return_bb, label %_loop_1_do_

_loop_1_do_:                              ; preds = %__m_NMOD_compute_entry, %_loop_1_do_
  %idx.03 = phi i32 [ %_loop_1_update_loop_ix, %_loop_1_do_ ], [ 1, %__m_NMOD_compute_entry ]
  %"_val_d-a1%addr_1" = load i32*, i32** getelementptr inbounds (%"_type_of_&&N&m", %"_type_of_&&N&m"* @"&&N&m", i64 0, i32 0), align 16
  %_val_a1_ = load i32, i32* %"_val_d-a1%addr_1", align 4
  %_val_b1_ = load i32, i32* getelementptr inbounds (%"_type_of_&&N&&m", %"_type_of_&&N&&m"* @"&&N&&m", i64 0, i32 0), align 16
  %"_val_d-a2%addr_3" = load i32*, i32** getelementptr inbounds (%"_type_of_&&N&m", %"_type_of_&&N&m"* @"&&N&m", i64 0, i32 6), align 16
  %_val_a2_ = load i32, i32* %"_val_d-a2%addr_3", align 4
  %_mult_tmp1 = add i32 %_val_a2_, %_val_a1_
  %_add_tmp = mul i32 %_mult_tmp1, %_val_b1_
  store i32 %_add_tmp, i32* %"_val_d-a1%addr_1", align 4
  %_loop_1_update_loop_ix = add nuw i32 %idx.03, 1
  %exitcond.not = icmp eq i32 %idx.03, %_val_n_
  br i1 %exitcond.not, label %_return_bb, label %_loop_1_do_

_return_bb:                               ; preds = %_loop_1_do_, %__m_NMOD_compute_entry
  ret void
}
```

Formalization

Consider a pair of IR instructions:

```
I1: %_val_1 = load i32, i32* %_load_addr_1, align 4
...
I2: store i32 %_val_2, i32* %_store_addr_2, align 4
```

Query: Does instruction `I2` write into the same memory that instruction `I1` loads from?

Objectives: To answer such queries in the following scenario

a frontend can compute definite alias relation between the instructions by applying language rules when a language-agnostic IR analyzer does not have sufficient information to determine the same.

Formalization

Problem to solve: To design an IR representation of arbitrary alias-graphs so that frontend can communicate language specific alias information to the analysis and transformation passes.

Option#1 for IR representation of alias-graph

To express alias relation between a given pair of instructions:

```
I1: %_val_1 = load i32, i32* %_load_addr_1, align 4, MayOperateOn={...}
...
I2: store i32 %_val_2, i32* %_store_addr_2, align 4, MayOperateOn={...}
```

Semantic:

MayOperateOn: A set of graph-nodes that this instruction *may* access (e.g. loads from, stores into, uses/modifies in a call) and any potential aliases of them.

[Query:] Does instruction `I2` write into the same memory that instruction `I1` loads from?

[**Answer:**]

"May" (i.e. mayAlias), in case,

MayOperateOn(`I1`) and MayOperateOn(`I2`) have *non-empty* intersection.

"No" (i.e. noAlias), otherwise.

Option#2 for IR representation of alias-graph

To express alias relation between a given pair of instructions:

```
I1: %_val_1 = load i32, i32* %_load_addr_1, align 4, OperatesOn={...} Unrelated={...}
...
I2: store i32 %_val_2, i32* %_store_addr_2, align 4, OperatesOn={...} Unrelated={...}
```

Semantic:

OperatesOn: A set of graph-nodes that this instruction accesses (e.g. loads from, stores into, uses/modifies in a call).

Unrelated: A set of graph-nodes that this instruction *provably does not* access.

[Query:] Does instruction `I2` write into the same memory that instruction `I1` loads from?

[**Answer:**]

"No" (i.e. noAlias), in case,

OperatesOn(`I1`) is a subset of Unrelated(`I2`) or

OperatesOn(`I2`) is a subset of Unrelated(`I1`)

"May" (i.e. mayAlias), otherwise.

Alias.scope-noalias based representation of option#2

- OperatesOn is represented by *alias.scope* metadata
- Unrelated is represented by *noalias* metadata

```
I1: %_val_1 = load i32, i32* %_load_addr_1, align 4, alias.scope={...} noalias={...}
                                                     OperatesOn={...} Unrelated={...}
...
I2: store i32 %_val_2, i32* %_store_addr_2, align 4, alias.scope={...} noalias={...}
                                                     OperatesOn={...} Unrelated={...}
```

Semantic:

OperatesOn: Each element in alias.scope set correpsonds to a graph-node that this instruction accesses.

Unrelated: Each element in noalias set corresponds to a graph-node that this instruction *provably does not* access.

[Query:] Does instruction `I2` write into the same memory that instruction `I1` loads from?

[**Answer:**] As computed by `ScopedNoAliasAAResult::alias`

"No" (i.e. noAlias), in case,

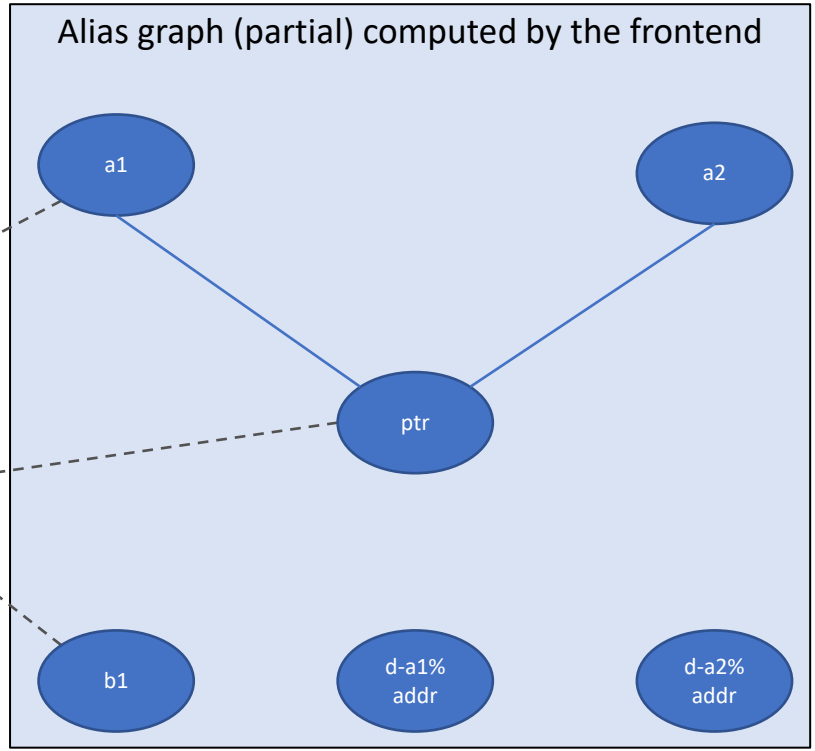
alias.scope(`I1`) is a subset of noalias(`I2`) or

alias.scope(`I2`) is a subset of noalias(`I1`)

"May" (i.e. mayAlias), otherwise.

IR representation of the alias graph

```
; Symbols (Graph nodes)
!1 = distinct !{!1, !2, !"init_var"}
!2 = distinct !{!2, !"domain: __m_NMOD__&&_m"}
!3 = distinct !{!3, !2, !"&&N&&m"}
!4 = distinct !{!4, !2, !"b1"}
!6 = distinct !{!6, !2, !"n"}
!7 = distinct !{!7, !2, !"idx"}
!9 = distinct !{!9, !2, !"d-a1%addr"}
!11 = distinct !{!11, !2, !"d-ptr%addr"}
!12 = distinct !{!12, !2, !"a1"}
!13 = distinct !{!13, !2, !"d-a2%addr"}
!14 = distinct !{!14, !2, !"a2"}
!15 = distinct !{!15, !2, !"ptr"}
!19 = distinct !{!19, !2, !"init_alloc_var"}
!22 = distinct !{!22, !2, !"__m_NMOD__&&_m"}
!23 = distinct !{!23, !2, !"__m_NMOD_init_ptr"}
!24 = distinct !{!24, !2, !"&&N&m"}
!25 = distinct !{!25, !2, !"__m_NMOD_compute"}
```



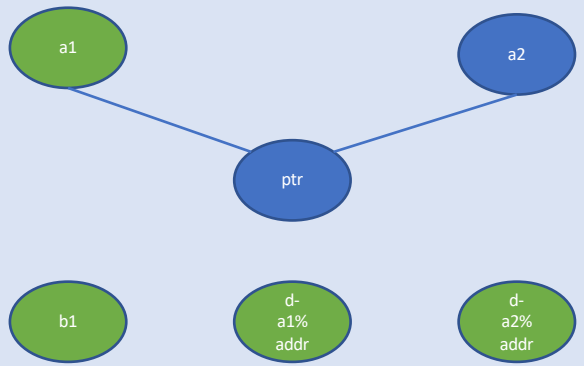
IR representation of the alias graph

```
; LOAD of a2
%_val_a2_ = load i32, i32* %"_val_d-a2%addr_3", align 4; OperatesOn={ a2 } Unrelated={ NoAlias set of a2 }

OperatesOn = { !14 }
Unrelated = {!3, !4, !6, !7, !9, !11, !12, !13, !24}
```

```
; Symbols (Graph nodes)
!1 = distinct !{!1, !2, !"init_var"}
!2 = distinct !{!2, !"domain: __m_NMOD__&&_m"}
!3 = distinct !{!3, !2, !"&&N&&m"}
!4 = distinct !{!4, !2, !"b1"}
!6 = distinct !{!6, !2, !"n"}
!7 = distinct !{!7, !2, !"idx"}
!9 = distinct !{!9, !2, !"d-a1%addr"}
!11 = distinct !{!11, !2, !"d-ptr%addr"}
!12 = distinct !{!12, !2, !"a1"}
!13 = distinct !{!13, !2, !"d-a2%addr"}
!14 = distinct !{!14, !2, !"a2"}
!15 = distinct !{!15, !2, !"ptr"}
!19 = distinct !{!19, !2, !"init_alloc_var"}
!22 = distinct !{!22, !2, !"__m_NMOD__&&_m"}
!23 = distinct !{!23, !2, !"__m_NMOD_init_ptr"}
!24 = distinct !{!24, !2, !"&&N&m"}
!25 = distinct !{!25, !2, !"__m_NMOD_compute"}
```

Alias graph (partial) computed by the frontend



Noalias set of a2

IR representation of the alias graph

```
; LOAD of a2
%_val_a2_ = load i32, i32* %"_val_d-a2%addr_3", align 4, !alias.scope !32, !noalias !33

; STORE into a1
store i32 %_add_tmp, i32* %"_val_d-a1%addr_1", align 4, !alias.scope !28, !noalias !29
```

!32 is a subset of !29: Load of a2 and store into a1 do not have any alias dependency.

```
; Symbols (Graph nodes)
!1 = distinct !{!1, !2, !"init_var"}
!2 = distinct !{!2, !"domain: __m_NMOD__&&_m"}
!3 = distinct !{!3, !2, !"&&N&&m"}
!4 = distinct !{!4, !2, !"b1"}
!6 = distinct !{!6, !2, !"n"}
!7 = distinct !{!7, !2, !"idx"}
!9 = distinct !{!9, !2, !"d-a1%addr"}
!11 = distinct !{!11, !2, !"d-ptr%addr"}
!12 = distinct !{!12, !2, !"a1"}
!13 = distinct !{!13, !2, !"d-a2%addr"}
!14 = distinct !{!14, !2, !"a2"}
!15 = distinct !{!15, !2, !"ptr"}
!19 = distinct !{!19, !2, !"init_alloc_var"}
!22 = distinct !{!22, !2, !"__m_NMOD__&&_m"}
!23 = distinct !{!23, !2, !"__m_NMOD_init_ptr"}
!24 = distinct !{!24, !2, !"&&N&m"}
!25 = distinct !{!25, !2, !"__m_NMOD_compute"}
```

```
; scope.alias sets
; (OperatesOn)
!0 = !{!1, !3, !4}
!8 = !{!9}
!16 = !{!13}
!18 = !{!19, !3,
        !12, !14}
!20 = !{!6}
!26 = !{!7}
!28 = !{!12}
!30 = !{!4}
!32 = !{!14}
!34 = !{!11}
!36 = !{!15}
```

```
; noalias sets (Unrelated)
!5 = !{!6, !7}
!10 = !{!7, !11, !3, !12, !13, !6, !4, !14, !15}
!17 = !{!7, !11, !3, !12, !6, !9, !4, !14, !15}
!21 = !{!7, !11, !3, !12, !13, !1, !22, !23, !19,
        !9, !4, !24, !14, !15, !25}
!27 = !{!11, !3, !12, !13, !1, !22, !23, !6, !19,
        !9, !4, !24, !14, !15, !25}
!29 = !{!7, !11, !3, !13, !6, !9, !4, !24, !14}
!31 = !{!7, !11, !12, !13, !6, !9, !24, !14, !15}
!33 = !{!7, !11, !3, !12, !13, !6, !9, !4, !24}
!35 = !{!7, !3, !12, !13, !6, !9, !4, !14, !15}
!37 = !{!7, !11, !3, !13, !6, !9, !4, !24}
```

IR after opt with alias information from frontend

```
Option: -O3 --unroll-max-count=0
define dso_local void @__m_NMOD_compute(i32* noalias nocapture readonly dereferenceable(4) %.n) local_unnamed_addr #1 {
…
_loop_1_do_.lr.ph:                                ; preds = %__m_NMOD_compute_entry
  %"_val_d-a1%addr_1" = load i32*, i32** getelementptr inbounds (%"_type_of_&&N&m", %"_type_of_&&N&m"* @"&&N&m", i64 0, i32 0), align 16,
!alias.scope !8, !noalias !10
  %_val_b1_ = load i32, i32* getelementptr inbounds (%"_type_of_&&N&&m", %"_type_of_&&N&&m"* @"&&N&&m", i64 0, i32 0), align 16, !alias.scope !26,
!noalias !27
  %"_val_d-a2%addr_3" = load i32*, i32** getelementptr inbounds (%"_type_of_&&N&m", %"_type_of_&&N&m"* @"&&N&m", i64 0, i32 6), align 16,
!alias.scope !16, !noalias !17
  %_val_a2_ = load i32, i32* %"_val_d-a2%addr_3", align 4, !alias.scope !28, !noalias !29
  %"_val_d-a1%addr_1.promoted" = load i32, i32* %"_val_d-a1%addr_1", align 4, !alias.scope !30, !noalias !31
  br label %_loop_1_do_

_loop_1_do_:                                      ; preds = %_loop_1_do_.lr.ph, %_loop_1_do_
  %_val_a1_4 = phi i32 [ %"_val_d-a1%addr_1.promoted", %_loop_1_do_.lr.ph ], [ %_add_tmp, %_loop_1_do_ ]
  %idx.03 = phi i32 [ 1, %_loop_1_do_.lr.ph ], [ %_loop_1_update_loop_ix, %_loop_1_do_ ]
  %_mult_tmp1 = add i32 %_val_a2_, %_val_a1_4
  %_add_tmp = mul i32 %_mult_tmp1, %_val_b1_
  %_loop_1_update_loop_ix = add nuw i32 %idx.03, 1
  %exitcond.not = icmp eq i32 %idx.03, %_val_n_
  br i1 %exitcond.not, label %_loop_1_loopHeader_._return_bb_crit_edge, label %_loop_1_do_

_loop_1_loopHeader_._return_bb_crit_edge:         ; preds = %_loop_1_do_
  store i32 %_add_tmp, i32* %"_val_d-a1%addr_1", align 4, !alias.scope !30, !noalias !31
  br label %_return_bb

_return_bb:                                       ; preds = %_loop_1_loopHeader_._return_bb_crit_edge, %__m_NMOD_compute_entry
  ret void
}
```

Run results with alias.scope-noalias representation

Execution performance

- No performance degradation for the workloads that we ran.
- Achieved moderate to significant improvement in execution performance for several workloads (2% - 200%).

	compile time increase	performance improvement	IR size increase
workload 1	1.8x	1x	1.4x
workload 2	4x	1.2x	3.2x
workload 3	15.5x	1x	2.5x
workload 4	6.6x	1.9x	8.7x

Desired attributes of IR representation of alias-graph

- Expressive power
  - Can the mechanism represent alias information with necessary preciseness?
- Compile-time performance
  - Do operations on alias sets such as insertion, update, concatenation, intersection, and query scale for large workloads?
- Memory footprint
  - Do the IR size and the size of in-memory representation scale for large workloads?

Attributes of alias.scope-noalias representation

- Expressive power
  - Can the mechanism represent alias information with necessary preciseness? Yes
- Compile-time performance
  - Do operations on alias sets such as insertion, update, concatenation, intersection, and query scale for large workloads? No, in the current implementation.
  - Keeping ordered set of MDOperands improves (amortized) the scalability of non-query operations.
  - Pre-partitioning the MDOperands improves the performance of the query operation.
- Memory footprint
  - Do the IR size and the size of in-memory representation scale for large workloads? No, in the current implementation.
  - Allowing hierarchical representation can improve the memory footprint. However, this may affect the query performance.

	compile time increase	performance improvement	IR size increase
workload 1	1.8x	1x	1.4x
workload 2	4x	1.2x	3.2x
workload 3	15.5x	1x	2.5x
workload 4	6.6x	1.9x	8.7x

Future work

- Investigate opportunities to improve the alias queries and intersections
- Explore opportunities to reduce the size of the alias.scope-noalias metadata
  - hierarchical representation
- Assess the impact of the full restrict patch (https://reviews.llvm.org/D69542)
  - perhaps some enhancements are needed
- How about a new alias representation for an arbitrary alias graph?

Active discussions in LLVM AA Technical Call every four weeks (https://docs.google.com/document/d/1ybwEKDVtIbhIhK50qYtwKsL50K-NvB6LfuBsfepBZ9Y/edit)

Any comments, feedback and collaboration are welcome

Backup

Impact of TBAA

```
struct N_m_t {
  struct d_type {
    int *addr;
    char dscr_type, data_type, flags, version;
  } d_a1, d_a2, d_ptr;
  int b1;
} N_m;

void init_var(int *);
void init_alloc_var(int **, int **);

void init_ptr() {
  *N_m.d_ptr.addr = 0;
}

void compute(int *n) {
  int idx;
  init var(&(N_m.b1));
  init_alloc_var(&N_m.d_a1.addr, &N_m.d_a2.addr);
  for (idx = 0; idx < *n; ++idx) {
    *N_m.d_a1.addr = *N_m.d_a1.addr * N_m.b1 + *N_m.d_a2.addr * N_m.b1;
  }
}
```

IR after opt with TBAA information from frontend

```
Option: -O3 --unroll-max-count=0

define dso_local void @compute(i32* nocapture readonly %0) local_unnamed_addr #1 {
...
  %2 = load i32, i32* %0, align 4, !tbaa !9
  %3 = icmp sgt i32 %2, 0
  br i1 %3, label %4, label %18

4:                                                ; preds = %1
  %5 = load i32*, i32** getelementptr inbounds (%struct.N_m_t, %struct.N_m_t* @N_m, i64 0, i32 0, i32 0), align 8, !tbaa !10
  %6 = load i32*, i32** getelementptr inbounds (%struct.N_m_t, %struct.N_m_t* @N_m, i64 0, i32 1, i32 0), align 8, !tbaa !11
  %7 = load i32, i32* %5, align 4, !tbaa !9
  br label %8

8:                                                ; preds = %4, %8
  %9 = phi i32 [ %7, %4 ], [ %14, %8 ]
  %10 = phi i32 [ 0, %4 ], [ %15, %8 ]
  %11 = load i32, i32* getelementptr inbounds (%struct.N_m_t, %struct.N_m_t* @N_m, i64 0, i32 3), align 8, !tbaa !12
  %12 = load i32, i32* %6, align 4, !tbaa !9
  %13 = add i32 %12, %9
  %14 = mul i32 %13, %11
  store i32 %14, i32* %5, align 4, !tbaa !9
  %15 = add nuw nsw i32 %10, 1
  %16 = load i32, i32* %0, align 4, !tbaa !9
  %17 = icmp slt i32 %15, %16
  br i1 %17, label %8, label %18

18:                                               ; preds = %8, %1
  ret void
}
```