

All About Alias Analysis

Nikita Popov

EuroLLVM 2026

Motivation

- Store to load forwarding (GVN, EarlyCSE)

<code>*x = 0;</code>		<code>*x = 0;</code>
<code>*y = 1;</code>		<code>*y = 1;</code>
<code>return *x;</code>	$\longrightarrow$	<code>return 0;</code>

Motivation

- Store to load forwarding (GVN, EarlyCSE)

<code>*x = 0;</code>		<code>*x = 0;</code>
<code>*y = 1;</code>		<code>*y = 1;</code>
<code>return *x;</code>	$\longrightarrow$	<code>return 0;</code>

- Do x and y alias?

Motivation

- Dead Store Elimination (DSE)

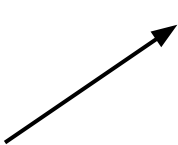
<code>*x = 0;</code>	$\longrightarrow$	
<code>*y;</code>		<code>*y;</code>
<code>*x = 1;</code>		<code>*x = 1;</code>

- Do x and y alias?

Motivation

- Loop Invariant Code Motion (LICM)

```
while (true) {  
    *x;  
    *y = 0;  
}  
  
while (true) {  
    *x;  
    *y = 0;  
}
```

A black arrow points from the first code block to the second. The arrow originates from the first code block and points towards the second code block, indicating a transformation or comparison between the two.

- Do x and y alias?

Agenda

- Memory accesses/locations/effects, provenance
- Aliasing and AA usage
- Alias analysis providers
- Derived analyses

Memory accesses

- load, store, call, ...
 - Not an access: getelementptr

Memory accesses

- load, store, call, ...
 - Not an access: getelementptr
- What does it access (MemoryLocation)?

Memory accesses

- load, store, call, ...
 - Not an access: getelementptr
- What does it access (MemoryLocation)?
- How does it access (ModRefInfo)?
 - NoModRef
 - Ref (read)
 - Mod (write)
 - ModRef (read+write)

MemoryLocation

- Base pointer
- Size
- Metadata (!tbaa, !noalias, etc.)

MemoryLocation

- Base pointer
- Size
- Metadata (!tbaa, !noalias, etc.)

LocationSize	Example
precise(n)	normal load/store

MemoryLocation

- Base pointer
- Size
- Metadata (!tbaa, !noalias, etc.)

LocationSize	Example
precise(n)	normal load/store
upperBound(n)	masked.load/store

MemoryLocation

- Base pointer
- Size
- Metadata (!tbaa, !noalias, etc.)

LocationSize	Example
precise(n)	normal load/store
upperBound(n)	masked.load/store
afterPointer()	memset with unknown size

MemoryLocation

- Base pointer
- Size
- Metadata (!tbaa, !noalias, etc.)

LocationSize	Example
precise(n)	normal load/store
upperBound(n)	masked.load/store
afterPointer()	memset with unknown size
beforeOrAfterPointer()	argument to unknown call

MemoryLocation

- Base pointer
- Size
- Metadata (!tbaa, !noalias, etc.)

LocationSize	Example
precise(n)	normal load/store
upperBound(n)	masked.load/store
afterPointer()	memset with unknown size
beforeOrAfterPointer()	argument to unknown call

Does this mean you can access **all** memory?

Provenance

- A pointer is more than an address

Provenance

- A pointer is more than an address
- Provenance: Virtual state that says what the pointer can access

Provenance

- A pointer is more than an address
- Provenance: Virtual state that says what the pointer can access
- Access outside the underlying object is undefined behavior

Provenance

- A pointer is more than an address
- Provenance: Virtual state that says what the pointer can access
- Access outside the underlying object is undefined behavior

```
unsigned x;  
unsigned y;  
if (&y == &x + 1) {  
    *(&x + 1) = 42;  
}
```

Provenance

- A pointer is more than an address
- Provenance: Virtual state that says what the pointer can access
- Access outside the underlying object is undefined behavior

```
unsigned x;  
unsigned y;  
if (&y == &x + 1) {  
    *(&x + 1) = 42; ← Undefined behavior  
}
```

MemoryEffects

- Calls may access more than one location
 - Can't describe with MemoryLocation

MemoryEffects

- Calls may access more than one location
 - Can't describe with MemoryLocation
- MemoryEffects: ModRefInfo per location kind
 - argmem
 - inaccessiblemem (not accessible via normal load/store)
 - errnomem
 - target_memN
 - other (globals, captured pointers, etc.)

MemoryEffects

- Calls may access more than one location
 - Can't describe with MemoryLocation
- MemoryEffects: ModRefInfo per location kind
 - argmem
 - inaccessiblemem (not accessible via normal load/store)
 - errnomem
 - target_memN
 - other (globals, captured pointers, etc.)
- Observable effects only
 - Read/write of alloca not observable outside call

MemoryEffects

- Examples:
 - `llvm.memcpy: memory(argmem: readwrite)`
 - `llvm.assume: memory(inaccessiblemem: readwrite)`

Aliasing

Def: Two locations alias if a memory access on one location can affect a memory access on the other location.

Aliasing

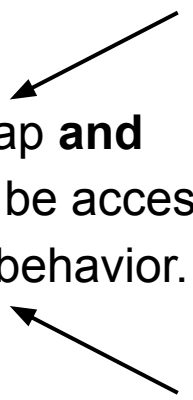
Def: Two locations alias if a memory access on one location can affect a memory access on the other location.

Two locations alias if

- the locations overlap **and**
- both locations can be accessed, with at least one access being a write, without undefined behavior.

Aliasing

Def: Two locations alias if a memory access on one location can affect a memory access on the other location.

- Two locations alias if
- the locations overlap **and**
 - both locations can be accessed, with at least one access being a write, without undefined behavior.
- Depends on address
- Depends on provenance
- 

Aliasing

```
unsigned x;  
unsigned y;  
if (&y == &x + 1) {  
    *(&x + 1) = 42; ← alias(&y, &x + 1) == NoAlias  
}
```

because access at &x + 1 is UB

alias(Loc1, Loc2)

- NoAlias
- MayAlias
 - Don't know
- MustAlias
 - Alias and start at same address
- PartialAlias
 - Alias, but may start at different addresses
 - Carries optional offset between base pointers

getModRefInfo()

- getModRefInfo(Instruction, Loc)
 - Can Instruction read/write Loc?

getModRefInfo()

- getModRefInfo(Instruction, Loc)
 - Can Instruction read/write Loc?
- getModRefInfo(Inst1, Inst2)
 - Can Inst1 read/write memory accessed by Inst2?
 - Excluding read-read cases.
 - Inst2 is usually a call.

-passes=aa-eval -print-all-alias-modref-info

```
define void @test(ptr %arg1, ptr %arg2) {  
    %alloca = alloca i32  
    store i32 1, ptr %alloca  
    store i32 2, ptr %arg1  
    store i32 3, ptr %arg2  
    call void @call()  
    ret void  
}
```

```
declare void @call()
```


-passes=aa-eval -print-all-alias-modref-info

```
define void @test(ptr %arg1, ptr %arg2) {  
    %alloca = alloca i32  
    store i32 1, ptr %alloca  
    store i32 2, ptr %arg1  
    store i32 3, ptr %arg2  
    call void @call()  
    ret void  
}  
  
declare void @call()
```

NoAlias: i32* %alloca, i32* %arg1
NoAlias: i32* %alloca, i32* %arg2
MayAlias: i32* %arg1, i32* %arg2

NoModRef: Ptr: i32* %alloca <-> call void @call()
Both ModRef: Ptr: i32* %arg1 <-> call void @call()
Both ModRef: Ptr: i32* %arg2 <-> call void @call()

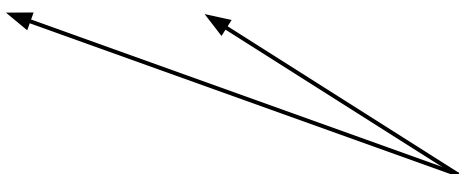
Cycles

```
void test(unsigned *p, size_t size) {  
    for (size_t i = 1; i < size; ++i) {  
        p[i] = p[i - 1] * 2;  
    }  
}
```

Cycles

```
void test(unsigned *p, size_t size) {  
    for (size_t i = 1; i < size; ++i) {  
        p[i] = p[i - 1] * 2;  
    }  
}
```

NoAlias ... on the same iteration!



Cycles

```
void test(unsigned *p, size_t size) {  
    for (size_t i = 1; i < size; ++i) {  
        p[i] = p[i - 1] * 2;  
    }  
}
```

NoAlias ... on the same iteration!

MayAlias in cross-iteration mode

Caching and BatchAA

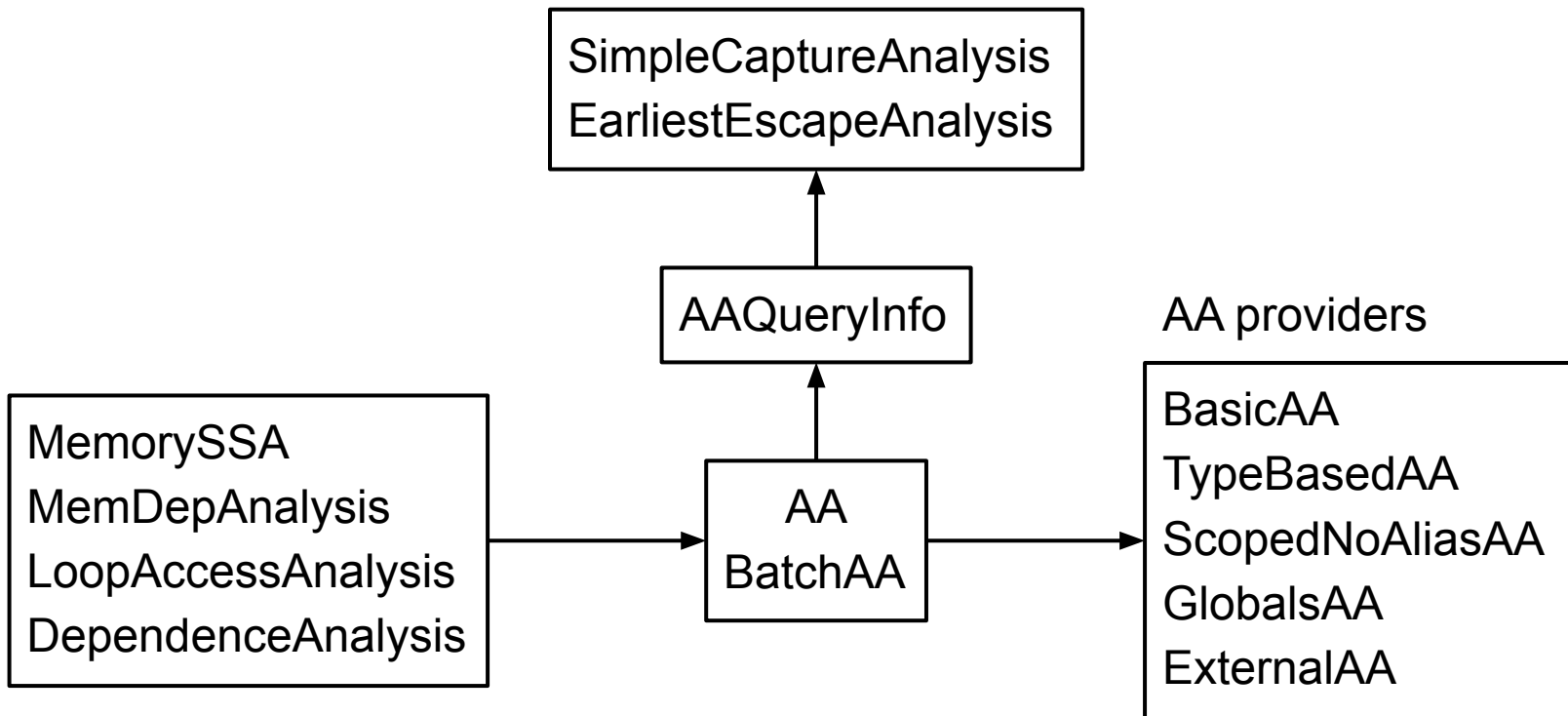
- AAQueryInfo contains cache (Loc1, Loc2) => AliasResult
- Normal AA uses separate AAQI per query
 - Caching only during recursive AA calls

Caching and BatchAA

- AAQueryInfo contains cache (Loc1, Loc2) => AliasResult
- Normal AA uses separate AAQI per query
 - Caching only during recursive AA calls
- BatchAA uses shared AAQI for all queries
 - Use whenever possible
 - No invalidation support

BatchAA: Erase XOR Insert

- Can use BatchAA across instruction erasure
- Can use BatchAA across instruction insert
- But not both
- Erase then insert may reuse pointer



AA providers

- BasicAA

AA providers

- BasicAA
- TypeBasedAA
 - Uses !tbaa metadata
 - Implements "strict aliasing" in C/C++

AA providers

- BasicAA
- TypeBasedAA
 - Uses !tbaa metadata
 - Implements "strict aliasing" in C/C++
- ScopedNoAliasAA
 - Uses !noalias and !alias.scope metadata
 - Representation of noalias after inlining (restrict in C, references in Rust)

AA providers

- BasicAA
- TypeBasedAA
 - Uses !tbaa metadata
 - Implements "strict aliasing" in C/C++
- ScopedNoAliasAA
 - Uses !noalias and !alias.scope metadata
 - Representation of noalias after inlining (restrict in C, references in Rust)
- GlobalsAA
 - Cached, interprocedural analysis of globals accesses

AA providers

- BasicAA
- TypeBasedAA
 - Uses !tbaa metadata
 - Implements "strict aliasing" in C/C++
- ScopedNoAliasAA
 - Uses !noalias and !alias.scope metadata
 - Representation of noalias after inlining (restrict in C, references in Rust)
- GlobalsAA
 - Cached, interprocedural analysis of globals accesses
- ExternalAA
 - Extension point: E.g. for address space aliasing on GPUs

BasicAA

- Recursive analysis based on pointer + size only (no metadata)
 - Recursion through phi, select, getelementptr

BasicAA

- Recursive analysis based on pointer + size only (no metadata)
 - Recursion through phi, select, getelementptr
- Implements:
 - Provenance based reasoning
 - Capture based reasoning
 - Offset based reasoning
 - ...

BasicAA: Provenance based reasoning

- `getUnderlyingObject()`
 - Strip off provenance-preserving operations like `getelementptr`
 - Offset / inbounds does not matter

BasicAA: Provenance based reasoning

- `getUnderlyingObject()`
 - Strip off provenance-preserving operations like `getelementptr`
 - Offset / inbounds does not matter
- `isIdentifiedObject()`: Does not alias with other identified objects
 - `alloca`
 - global variable
 - call with `noalias` return (allocator)
 - `noalias` or `byval` argument

BasicAA: Provenance based reasoning

- $\text{Obj1} \neq \text{Obj2} \ \&\& \ \text{isIdentifiedObject}(\text{Obj1}) \ \&\& \ \text{isIdentifiedObject}(\text{Obj2})$
 - $\Rightarrow \text{NoAlias}$
 - For example: Two allocas don't alias

BasicAA: Provenance based reasoning

- `Obj1 != Obj2 && isIdentifiedObject(Obj1) && isIdentifiedObject(Obj2)`
 - `=> NoAlias`
 - For example: Two allocas don't alias
- `Obj1 != Obj2 && isa<Argument>(Obj1) && isIdentifiedFunctionLocal(Obj2)`
 - `=> NoAlias`
 - `isIdentifiedFunctionLocal()` excludes globals
 - For example: An argument and an alloca don't alias

BasicAA: Capture based reasoning

```
%a = alloca i32  
call void @escape_ptr(ptr %a)  
%b = call ptr @get_ptr()
```

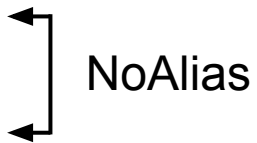


MayAlias

BasicAA: Capture based reasoning

```
%a = alloca i32
```

```
%b = call ptr @get_ptr()
```



Escape source

- Can only alias with previously escaped object
 - call return value (with some exceptions)
 - load
 - inttoptr
 - extractvalue, extractelement

Is it captured before?

```
void test() {  
    unsigned x;  
    *x;  
    if (cond) {  
        *x;  
        escape(&x);  
    } else {  
        escape(&x);  
    }  
    *x;  
}
```

Is it captured before?

```
void test() {  
    unsigned x;  
    *x; ←  
    if (cond) {  
        *x; ←  
        escape(&x);  
    } else {  
        escape(&x);  
    }  
    *x; ←  
}
```


Is it captured before?

```
void test() {  
    unsigned x;                                Simple  
    *x; ←──────────────────────────────── yes  
    if (cond) {  
        *x; ←──────────────────────── yes  
        escape(&x);  
    } else {  
        escape(&x);  
    }  
    *x; ←──────────────────────── yes  
}
```

Is it captured before?

```
void test() {  
    unsigned x;                                Simple  
    *x; ←──────────────────────────────── yes    CapturesBefore  
    if (cond) {  
        *x; ←──────────────────────────────── yes    no  
        escape(&x);  
    } else {  
        escape(&x);  
    }  
    *x; ←──────────────────────────────── yes    yes  
}
```

Is it captured before?

```
void test() {  
    unsigned x;  
    *x; ← Simple yes  
    if (cond) { ← EarliestEscape  
        *x; ← Nearest common dominator of escapes  
        escape(&x);  
    } else {  
        escape(&x);  
    }  
    *x; ← CapturesBefore yes  
}
```

Is it captured before?

		Simple	EarliestEscape	CapturesBefore
void test() {				
unsigned x;				
*x;	←	yes	no	no
if (cond) {	← Nearest common dominator of escapes			
*x;	←	yes	yes	no
escape(&x);				
} else {				
escape(&x);				
}				
*x;	←	yes	yes	yes
}				

CaptureAnalysis

- SimpleCaptureAnalysis
 - Not context sensitive
 - Used by default

CaptureAnalysis

- SimpleCaptureAnalysis
 - Not context sensitive
 - Used by default
- EarliestEscapeAnalysis
 - Cached nearest common dominator of escapes
 - Supports invalidation

BasicAA: Offset based reasoning

- Reasons about access overlap (provenance not relevant)
- Decompose chain of getelementptrs (and indices)
 - $\text{BasePtr} + \text{ConstOffset} + \text{Scale1} * \text{Var1} + \dots$

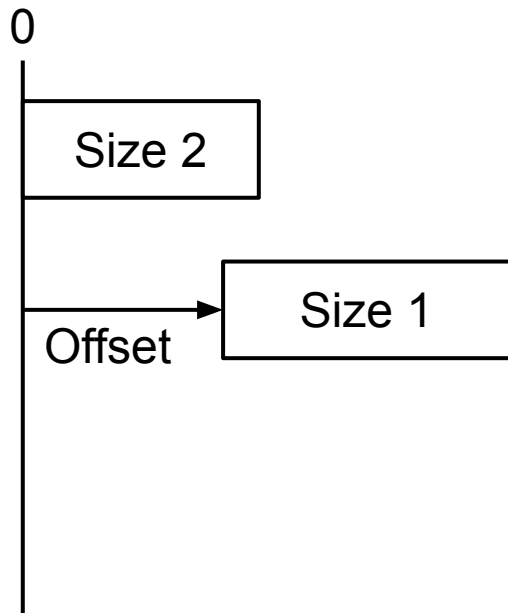
BasicAA: Offset based reasoning

- Reasons about access overlap (provenance not relevant)
- Decompose chain of getelementptrs (and indices)
 - $\text{BasePtr} + \text{ConstOffset} + \text{Scale1} * \text{Var1} + \dots$
- Two decomposed GEPs with common base:
 - Access 1 at $\text{BasePtr} + \text{Offset1}$
 - Access 2 at $\text{BasePtr} + \text{Offset2}$

BasicAA: Offset based reasoning

- Reasons about access overlap (provenance not relevant)
- Decompose chain of getelementptrs (and indices)
 - $\text{BasePtr} + \text{ConstOffset} + \text{Scale1} * \text{Var1} + \dots$
- Two decomposed GEPs with common base:
 - Access 1 at $\text{BasePtr} + \text{Offset1}$
 - Access 2 at $\text{BasePtr} + \text{Offset2}$
- Subtract $\text{BasePtr} + \text{Offset2}$ from both:
 - Access 1 at $\text{Offset1} - \text{Offset2}$
 - Access 2 at 0

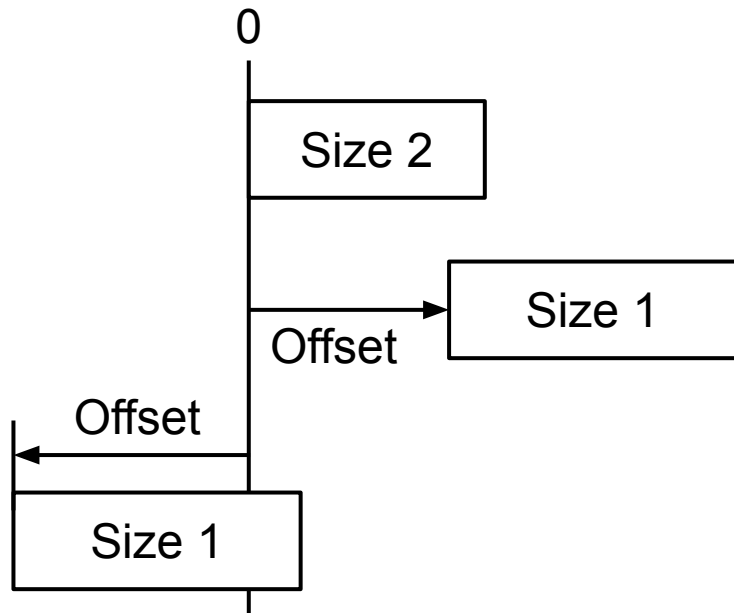
BasicAA: Offset based reasoning



NoAlias if:

Offset $\geq$ Size 2

BasicAA: Offset based reasoning



NoAlias if:

$\text{Offset} \geq \text{Size } 2$

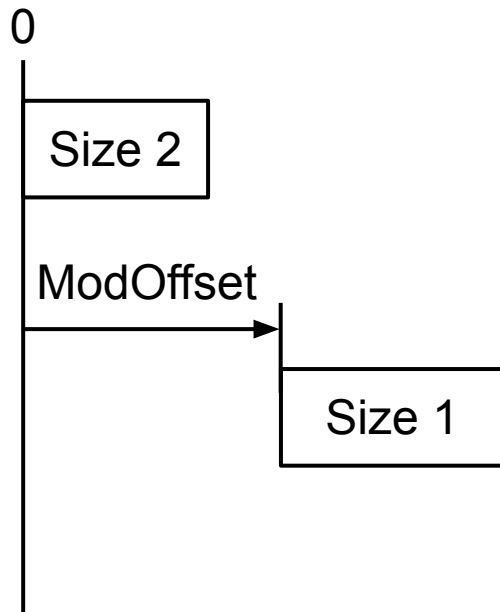
$-\text{Offset} \geq \text{Size } 1$

BasicAA: GCD heuristic

$\text{Offset} = \text{ModOffset} + \text{GCD} * X$ (where $\text{ModOffset} < \text{GCD}$)

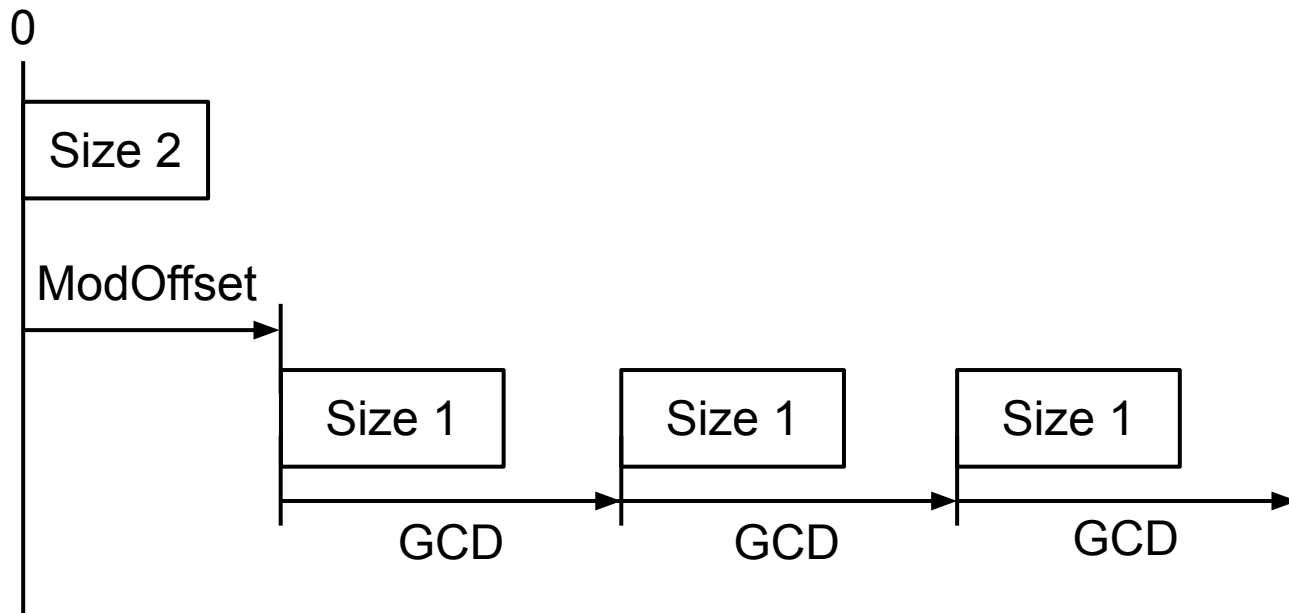
BasicAA: GCD heuristic

$\text{Offset} = \text{ModOffset} + \text{GCD} * X$ (where $\text{ModOffset} < \text{GCD}$)



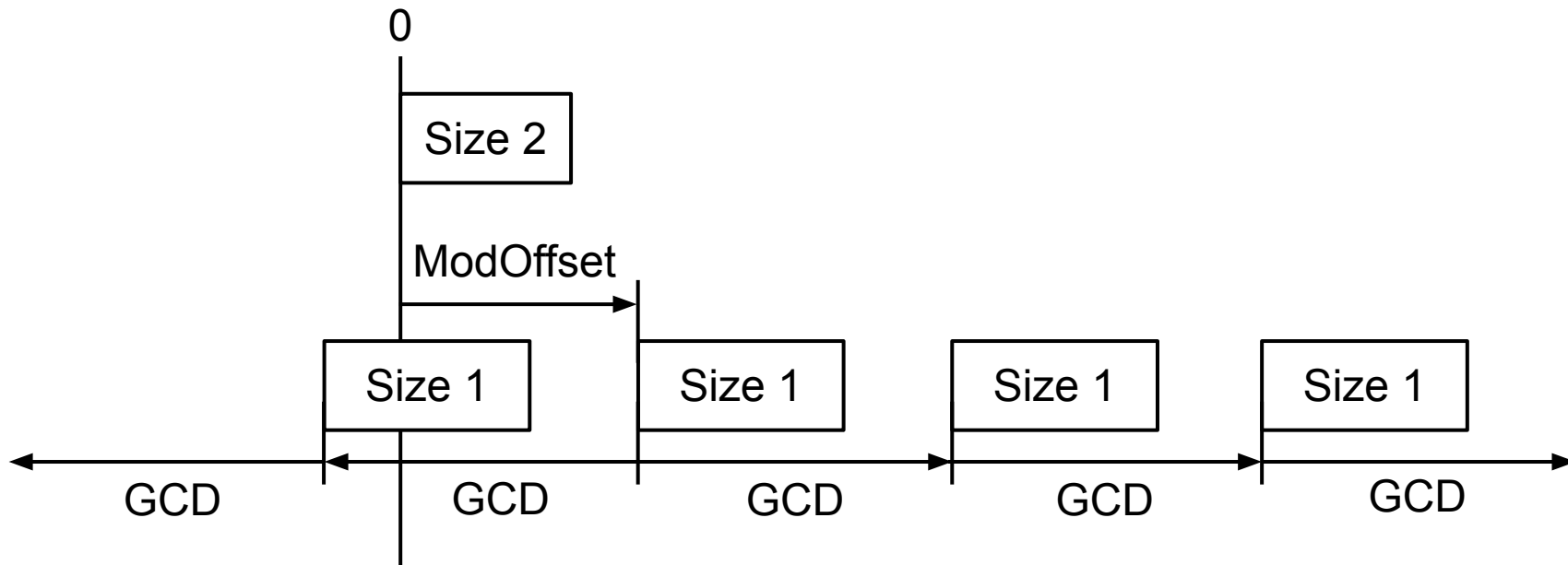
BasicAA: GCD heuristic

$\text{Offset} = \text{ModOffset} + \text{GCD} * X$ (where $\text{ModOffset} < \text{GCD}$)



BasicAA: GCD heuristic

Offset = ModOffset + GCD*X (where ModOffset < GCD)

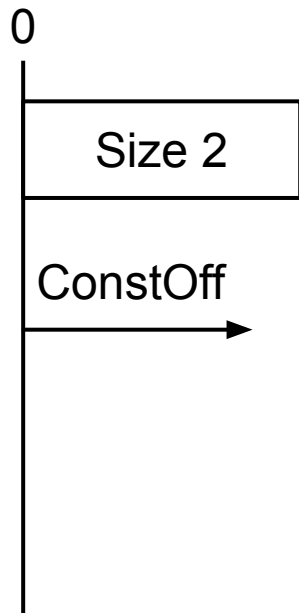


BasicAA: Min abs heuristic

Offset = ConstOff + Scale * X, where $X \neq 0$

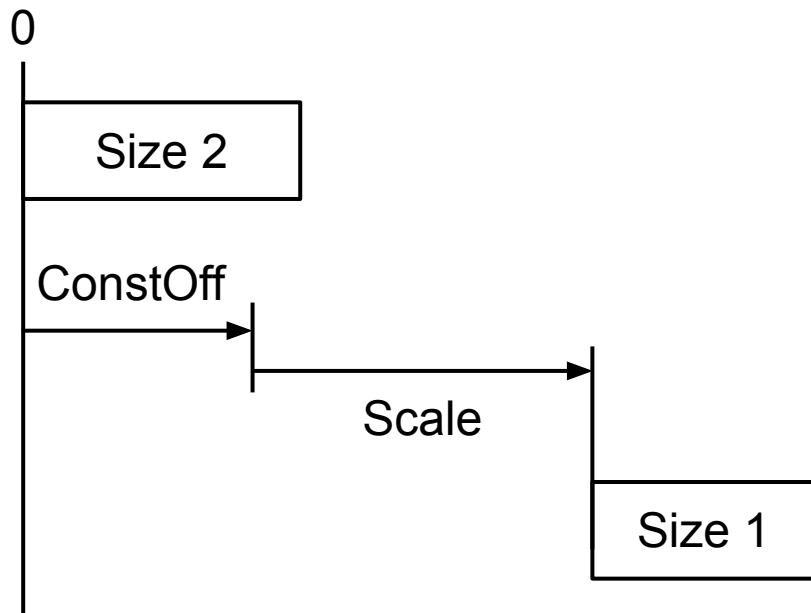
BasicAA: Min abs heuristic

Offset = ConstOff + Scale * X, where $X \neq 0$



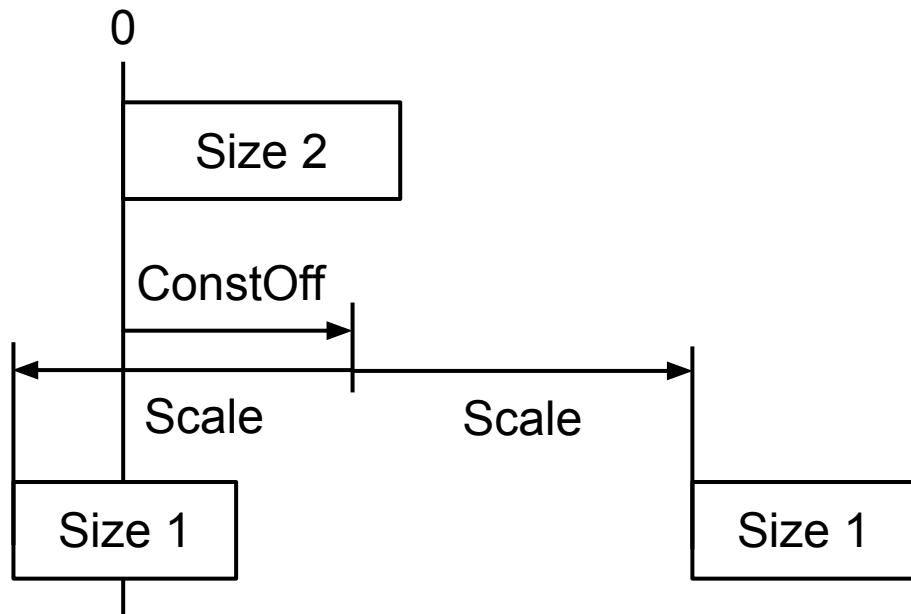
BasicAA: Min abs heuristic

Offset = ConstOff + Scale * X, where $X \neq 0$



BasicAA: Min abs heuristic

Offset = ConstOff + Scale * X, where $X \neq 0$



BasicAA: Offset based reasoning

- Real implementation is more complicated
- Reason: Overflow
- Careful tracking of inbounds, nuw, nsw and negations

Derived analyses

- MemorySSA

Derived analyses

- MemorySSA
- MemDepAnalysis
 - Obsoleted by MemorySSA (but still used by GVN)

Derived analyses

- MemorySSA
- MemDepAnalysis
 - Obsoleted by MemorySSA (but still used by GVN)
- LoopAccessAnalysis (LAA)
 - Reasons about dependences inside loops
 - Can generate runtime checks if not statically provable
 - Main user: Loop vectorization

Derived analyses

- MemorySSA
- MemDepAnalysis
 - Obsoleted by MemorySSA (but still used by GVN)
- LoopAccessAnalysis (LAA)
 - Reasons about dependences inside loops
 - Can generate runtime checks if not statically provable
 - Main user: Loop vectorization
- DependenceAnalysis
 - Like LAA, but can reason about nested loops / multi-dimensional accesses
 - No current users in default pipeline
 - Work in progress to fix correctness issues

MemorySSA

- Memory versioning
- $\text{NewVersion} = \text{MemoryDef}(\text{PreviousVersion})$
- $\text{NewVersion} = \text{MemoryPhi}(\{\text{block}, \text{PreviousVersionInBlock}\}, \dots)$
- $\text{MemoryUse}(\text{DepVersion})$

MemorySSA

```
define void @test(i1 %c, ptr %p) {  
    entry:  
        %a = alloca i32, align 4  
        ; 1 = MemoryDef(liveOnEntry)  
        store i32 1, ptr %a, align 4  
        br i1 %c, label %if, label %join  
  
    if:  
        ; 2 = MemoryDef(1)  
        store i32 0, ptr %p, align 4  
        br label %join  
  
    join:  
        ; 3 = MemoryPhi({entry,1},{if,2})  
        ; MemoryUse(3)  
        %load.p = load i32, ptr %p, align 4  
        ; MemoryUse(3)  
        %load.a = load i32, ptr %a, align 4  
        ret void  
}
```

```
; -passes='print<memoryssa><no-ensure-optimized-uses>'
```

MemorySSA

```
define void @test(i1 %c, ptr %p) {  
    entry:  
        %a = alloca i32, align 4  
        ; 1 = MemoryDef(liveOnEntry)  
        store i32 1, ptr %a, align 4  
        br i1 %c, label %if, label %join  
  
    if:  
        ; 2 = MemoryDef(1)  
        store i32 0, ptr %p, align 4  
        br label %join  
  
    join:  
        ; 3 = MemoryPhi({entry,1},{if,2})  
        ; MemoryUse(3)  
        %load.p = load i32, ptr %p, align 4  
        ; MemoryUse(1)  
        %load.a = load i32, ptr %a, align 4  
        ret void  
}
```

```
; -passes='print<memoryssa>'
```

MemorySSA clobber walker

- `getClobberingMemoryAccess(MemoryAccess)`
 - Find dominating clobber of any memory read/written by the access (excluding access)
 - Cached
- `getClobberingMemoryAccess(MemoryAccess, MemoryLocation)`
 - Find dominating clobber of given memory location (including access)

MemorySSA clobber walker

- `getClobberingMemoryAccess(MemoryAccess)`
 - Find dominating clobber of any memory read/written by the access (excluding access)
 - Cached
- `getClobberingMemoryAccess(MemoryAccess, MemoryLocation)`
 - Find dominating clobber of given memory location (including access)
- `-passes='print<memoryssa-walker>'`

Future directions

- Support encoding more aliasing information in IR
 - Full restrict patches
- Improve compile-time
 - And/or our use of the compile-time budget
- Better reasoning about loop-varying pointers
 - SCEVAA exists, but not used. Expensive.

Thank You!

Questions?

