Finding Your Way Around the LLVM Dependence Analysis Zoo

MemorySSA and DependenceAnalysis Tutorial
Outline

- What is Dependence Analysis? Why do we care?
- Basic Theory
- MemorySSA, DependenceAnalysis:
  - What are they?
  - Theoretical Foundation
  - Important Implementation Details
  - Understanding their Output
Why Do We Care About Dependence Analysis?

In reordering transformations, preserve the dependences and you preserve the semantics!
What Is Dependence Analysis?

Gathering information about the dependences of a program.
Example: Read-After-Write (RAW)

```cpp
1 int x = 2;
2 int y = 3;
3 int c = x * y;
```
Example: Write-After-Read (WAR)

1 // x == 10
2 int y = x * 2;
3 x = 3;
Example: Write-After-Write (WAW)

1 int x = 10;
2 x = 20;
3 int c = x * 2;
What is a Dependence?

- Dependence is an ordering between two operations that we have to preserve.
- This arises because if we don’t, a *read* may break.
- A *data* dependence exists because the two operations access the same memory location.
MemorySSA
Why MemorySSA?

- Clean theory
- Minimalistic interface
- Actively used & maintained
The Idea
%x = add %a, %b
%y = mul %x, %c
%z = sub %x, %y
Def-Use Chains

%\(x\) = add \(a, b\)
%\(y\) = mul \(x, c\)
%\(z\) = sub \(x, y\)
Def-Use Chains

\[
\begin{align*}
\%x &= \text{add} \ %a, \ %b \\
\%y &= \text{mul} \ %x, \ %c \\
\%z &= \text{sub} \ %x, \ %y
\end{align*}
\]

```cpp
llvm::Value *X = /* %x */
for (auto *User : X->users()) {
    print(*User)
}
// %y, %z

llvm::Instruction *Z = /* %z */
for (auto *Op : Z->operands()) {
    print(*Op)
}
// %x, %y
```
dependence

\[
\begin{align*}
\text{store } \%v, \ i32^* \ %a \\
\%y = \text{load } i32^* \ %b \\
\%z = \text{load } i32^* \ %c
\end{align*}
\]

\[
\text{llvm::Instruction *Z = /* } \%z \ */ \\
\text{for (auto *Op : Z->operands())} \ \{ \\
\text{print(*Op)} \\
\} \\
\]

// %c %y, %z
Clobber & Alias

store %v, i32* %a
%y = load i32* %b
%z = load i32* %c

**Alias:** Can %c point to the same memory as %a?

**Clobber:**
If a store happens before a load and the pointers alias.
-> the store is a clobber of the load
Clobber & Alias

Alias: Can %c point to the same memory as %a?

Clobber:
If a store happens before a load and the pointers alias.
-> the store is a clobber of the load

store %v, i32* %a
%y = load i32* %b
%z = load i32* %c
SSA on versioned Memory

- **liveOnEntry** - memory state at function entry

- **x = MemoryDef(y)** - modify memory version \( y \) producing \( x \) (eg for a store)

- **MemoryUse(x)** - read memory version \( x \) (eg for a load)

- **MemoryPhi(x,y,..)** - merge incoming memory versions at block entry
store %v, i32* a

%y = load i32* %b

%z = load i32* %c

0=liveOnEntry

MemoryDef

MemoryUse

MemoryUse
%a = alloca i32
%b = alloca i32
%c = alloca i32

store %v, i32* %a

%y = load i32* %b

%z = load i32* %c
def @foo(i32* %a, i32 %i) {
    %b = getelementptr %a, %i
    %c = alloca i32
    store %v, i32* %a

    %y = load i32* %b

    %z = load i32* %c
}
Memory SSA

```c
define void @f(i32* %p, i1 %cond) {
  entry:
    ; MemoryUse(liveOnEntry)
    %0 = load i32, i32* %p, align 4
    br i1 %cond, label %if.then, label %if.end

  if.then:
    ; 1 = MemoryDef(liveOnEntry)
    store i32 42, i32* %p, align 4
    br label %if.end

  if.end:
    ; 2 = MemoryPhi({entry, liveOnEntry}, {if.then, 1})
    ; MemoryUse(2)
    %1 = load i32, i32* %p, align 4
    ret void
}
```
Limitations

..and how to walk past them
def @foo(i32* noalias A, i32* noalias B) {
    ...
    store i32 1, i32* %A
    ...
    store i32 2, i32* %B
    ...
    store i32 3, i32* %A
    ...
    store i32 4, i32* %B
    ...
}
def @foo(i32* noalias A, i32* noalias B) {
    ... 
    store i32 1, i32* %A 
    ... 
    store i32 2, i32* %B 
    ... 
    store i32 3, i32* %A 
    ... 
    store i32 4, i32* %B 
    ... 
}

(not actually the Memory SSA graph)
def @foo(i32* noalias A, i32* noalias B) {
    ...
    store i32 1, i32* %A
    ...
    store i32 2, i32* %B
    ...
    store i32 3, i32* %A
    ...
    store i32 4, i32* %B
    ...
}
The Walker

def @foo(i32* noalias A, i32* noalias B) {
    ...
    store i32 1, i32* %A
    ...
    store i32 2, i32* %B
    ...
    store i32 3, i32* %A
    ...
    store i32 4, i32* %B
    ...
}

auto *Walker = MemorySSA->getWalker();
Walker->getClobberingMemoryAccess(/* 4 */)
// 2 = MemoryDef(1)
Conclusion

- MemorySSA: SSA on memory versions.
- Better results with The Walker.
- Use it! Clean, maintained, actively used, evolving
Stuff I didn’t talk about

- How does MemorySSA know what aliases what?
  - \(\rightarrow\) AliasAnalysis

- Custom Walkers

- MayAlias, MustAlias, ModRef, ..
DependenceAnalysis analyzes dependences between pairs of memory accesses. Currently, it is an (incomplete) implementation of the approach described in:

Practical Dependence Testing
Goff, Kennedy, Tseng
PLDI 1991

```cpp
std::unique_ptr<Dependence>
DependenceInfo::depends(Instruction *Src,
                        Instruction *Dst,
                        bool PossiblyLoopIndependent) {
```
Loops Are Especially Interesting

Loop-Specific Dependences:
- Loop-Independent
- Loop-Carried
Example: Loop-Independent Dependence

```c
for (int i = 0; i < ...; ++i) {
    A[i] *= 2;
    y += A[i] + C;
}
```

Any single iteration of the loop has this dependence.
Example: Loop-Carried Dependence

for (int i = 0; i < ...; ++i) {
    int temp = A[i];
    A[i + 2] = temp;
}

Exists exactly because of the loop. One iteration depends on another.
Example: Loop-Carried Dependence (Unrolled)

Statements in lines 3 and 8 are dependent.

```
1 i = 0;
2 temp = A[0];
4 i = 1;
5 temp = A[1];
7 i = 2;
8 temp = A[2];
10 ...
```
Distance / Direction Vectors

```c
for (int i = 0; i < ...; ++i) {
    int temp = A[i];
    A[i + 2] = temp;
}
```

How many iterations from one access to another (on the same memory location)?
Example: Dependence Distance

The distance is (usually) constant.
Multi-Dimensional Distance / Direction Vectors

```cpp
for (int i = 0; i < ...; ++i)
    for (int j = 0; j < ...; ++j)
        for (int k = 0; k < ...; ++k)
            A[i+1][j][k-1] = A[i][j][k] + C
```

Distance Vector: (1, 0, -1)
Direction Vector: (<, =, >)
Dependence Tests

How can the compiler deduce (in)dependences in some automatic, yet precise way?
Indices and Subscripts

1 for (int i = 0; i < ...; ++i)
2 for (int j = 0; j < ...; ++j)
3 for (int k = 0; k < ...; ++k)
4 A[i][j] = A[i][k];

Indices of the loop nest: i, j, k
Subscripts of the access pair: (i, i), (j, k)
Subscript Classification

1) Complexity
2) Separability
Subscript Complexity

1  // Assume that `N` is loop-invariant.
2  for (int i = 0; i < ...; ++i)
3      for (int j = 0; j < ...; ++j)
4          for (int k = 0; k < ...; ++k)

How many indices each subscript uses?
Subscript Separability

1 // Assume that `N` is loop-invariant.
2 for (int i = 0; i < ...; ++i)
3  for (int j = 0; j < ...; ++j)
4     for (int k = 0; k < ...; ++k)
5         A[i][j][j] = A[i][j][k] + C;

How many subscripts use the same index?
This is all good but...

LLVM IR does not have indices, subscripts or C-style multi-dimensional array accesses.

Quick answer: SCEV everywhere.
Multi-dimensional accesses in C: Multi-Indirection Pointers

```c
1 int ***A;
2 ...
3 A[i][j][k];
```

Difficult to deal with because of no aliasing guarantees.
Multi-dimensional accesses in C: 
“Multi-Dimensional” Arrays

1 `int A[][][M];`
2 ...
3 `A[i][j] is really A[i*M + j]`

A multi-dimensional access is just syntactic sugar for a linear access.
Multi-dimensional accesses in C: “Multi-Dimensional” Arrays

We have to use SCEV Delinearization to turn $A[i*M + j]$ back to $A[i][j]$, which is not always perfect.
Multi-dimensional accesses in C: “Multi-Dimensional” Arrays

1 int A[][M];
2 ...
3 A[i][j] is really A[i*M + j]

Because it’s actually a linear access, there are no in-bounds guarantees for each dimension.
Returning to our question: How do we come up with automatic dependence tests?

Quick answer: Look at the subscripts.
ZIV (Zero Index Variable) Test

No indices used in the subscript. Two cases: They’re either equal or they’re not.
They are equal. We can squash their dimension.
ZIV (Zero Index Variable) Test

1 for (int i = 0; i < ...; ++i)
2    for (int j = 0; j < ...; ++j)
3        A[i] = A[i+1];

Equivalent subscripts.
ZIV (Zero Index Variable) Test

1 for (int i = 0; i < ...; ++i)
2 for (int j = 0; j < ...; ++j)
3 A[i][0] = A[i][1];

They’re *not* equal. We always access different columns, so no dependence.
The ZIV subscripts can be complex as long as they’re loop-nest-invariant.
SIV (Single Index Variable) Subscript Test

Exactly one index used in the subscript. Hard to solve in full generality. We show 2 common subcases.
Strong SIV Test:
\[(a_i + c_1, a_i + c_2)\]

```
1 for (\text{int} \ i = 0; i < 2*N; i += 2)
2   A[i+3] = A[i];
3
4 -->
5
6 for (\text{int} \ i = 0; i < N; ++i)
8
9 Subscript: (2*i + 3, 2*i)
10 a = 2, c_1 = 3, c_2 = 0
```

\(a\) is usually the step.
Strong SIV Test: 
\((ai + c1, ai + c2)\)

Dependence Distance: 
\[d = \frac{c1 - c2}{a}\]

You have to cover \(c1 - c2\) distance by moving in steps of \(a\).
Strong SIV Test: 
\((a_i + c_1, a_i + c_2)\)

Dependence Distance: 
\[ d = \frac{c_1 - c_2}{a} \]

A dependence exists if and only if \(d\) is an integer and \(|d| \leq U - L\), where \(U\) and \(L\) are the loop upper and lower bounds.
Weak SIV Subscripts:
\[(a_1i + c_1, a_2i + c_2)\]

Now \(a_1 \neq a_2\) ! Again, it’s hard to solve it in full generality but we show 2 common subcases.
Weak-Zero SLV Subscripts:
\((a1*i + c1, a2*i + c2)\)

Subcases:
- (Weak-Zero) \(a1 = 0\) or \(a2 = 0\)
- (Weak-Crossing) \(a1 = -a2\)
Weak-Zero SIV Test:

\((a_1i + c_1, a_2i + c_2)\)

\(a_1 = 0\) or \(a_2 = 0\). Assume \(a_2 = 0\).

It finds dependences caused by a particular iteration \(i = \frac{c_2 - c_1}{a_1}\)

Again, \(i\) needs to be an integer and within loop bounds for a dependence to exist.
Weak-Zero SIV Test:

\((a_1*i + c_1, a_2*i + c_2)\)

\(A[1][N]\) causes a dependence from the first iteration to all others. Similarly, \(A[N][N]\) causes a dependence from all iterations to the last. We can peel those two!

```c
1    for (int i = 1; i <= N; ++i) 
```
Peel the first and last iterations

2 \[ \text{for} \ (\text{int} \ i = 2; \ i \leq N-1; \ ++i) \]
Weak-Crossing SIV Test:

\((a_1i + c_1, a_2i + c_2)\)

\[a_1 = -a_2.\] It finds dependences meeting at a particular iteration:

\[i = \frac{c_2 - c_1}{2a_1}\]

Why 2 is there? And what the condition for a dependence is?
Weak SIV Subscripts:

\((a_1i + c_1, a_2i + c_2)\)

In general, we can view the SIV tests as line tests.
Geometric View of SIV Tests

- Bounded Iteration Space
- Strong SIV
- Weak-Zero SIV
- Weak-Crossing SIV

DO 10 i = 1,4
10 A(i) = A(i-1)

DO 10 i = 1,4
10 A(i) = A(1)

DO 10 i = 1,4
10 A(i) = A(5-i)