How LLVM does Memory Dependence Analysis

Introduction

In LLVM, the memory dependence analysis computes the memory dependence information for a function. Consider the following piece of code inside a function:

```
1: int *a = malloc(sizeof(int);
2: *a = 5;
3: printf("a is %d\n", a);
```

There is a write->read dependence from line 2 to line 3. The memory dependence analysis in LLVM captures exactly that.

Even if the analysis is lazy (on demand) in computation, and the result will be cached, the analysis is still very costly. A query of memory depedence for a specific instruction may takes up to quadratic time. In LLVM 7.0.0, there is a recent attempt to introduce `memorySSA`, which is an upgraded SSA form that can reason about the interactions between various memory operations, to replace the memory dependence analysis.

Version: LLVM 7.0.0

Author: Ziang Wan

Memory Dependence Analysis: the Pass -memdep

To tell LLVM to run the analysis pass for your code:

```
opt -memdep ...
```

If your pass needs to get the memory dependence information for the current function, you should use:

```
// in your getAnalysisUsage(AnalysisUsage &AU)
AU.addRequired<MemoryDependenceWrapperPass>();
...
// later in your code
MemoryDependenceResults &MDR = getAnalysis<MemoryDependenceWrapperPass>().getMemDep();
```

Source Code Locations

- /lib/Analysis/MemoryDependenceAnalysis.cpp
- /include/llvm/Analysis/MemoryDependenceAnalysis.h

About the MemoryDependenceResults object The pass will give you a `MemoryDepenceResults` object that serves as an interface for obtaining memory dependence informations for the current function. The object is also a cache for these informations in order to avoid costly re-computation. In other words, the `-memdep` pass is nearly a no-op and simply gives you an interface. As you query for what preceding memory operations a specific instruction depends on, the object will compute, cache and return them to you on the fly.

Internally, the cache of MemoryDependenceResults is represented using a map that maps each instruction in the function to those on which it depends on. Functions such as `MemoryDependenceResults::removeInstruction` and `MemoryDependenceResults::invalidateCachedPredecessors` (when the CFG structure changes) will invalidate the cache.

For each memory operation instruction that a specific instruction depends on, it is classified into three tyeps:

- local memory dependence within the basic block
- nonlocal memory dependence within the function but outside the basic block
- nonFuncLocal memory dependence that is both outside the basic block and the function

The analysis does not concern with nonFuncLocal memory dependence. The analysis does not detect memory operations outside the function and only returns instructions inside the function. To get local memory dependence for an instruction: `MemoryDependenceResults::getDependency`. To get nonlocal memory dependence for an instruction: `MemoryDependenceResults::getNonLocalCallDependency` (for function call) and `MemoryDependenceResults::getNonLocalPointerDependency`.

For each memory dependence relationship from one instruction to another instruction inside a function, it is further classified into two types. Each type has specific cases of interest: * Clobber * from a may-alias store to a load * from a partially-alised load to a load * Def * from a must-alias load to a store * from a must-alias store to a load * from an allocation to a load / a store * from a must-alias store to a call site

In summary, there are 12 types of memory dependence relationships that can be detected by LLVM's memory dependence analysis through a `MemoryDependenceResults` object.

The algorithm & its Time Complexity There are a total 12 cases to study. The 12 cases differ in the algorithms they use. For example, when computing local memory dependence information for a store/load, the analysis simply walks through the local basic block and collects all the instructions that satisfies either the Clobber or the Def criteria. When computing local memory dependence

information for a call site, the memory aliasing information for that call site is also used.

Computing the memory dependence relationship for an instruction takes up to O(N) time, not considering the alias analysis part, where N is the number of instructions in the function. Therefore, comptuing the memory dependence information for a function takes up to O(N^2) time, where N is the number of instructions in the function. You won't necessarily reach O(N^2) because the computation is lazy, and you won't necessarily need the memory dependence information for every instruction in the function.

I will show you a brief example of the memory dependence analysis for a simple c program:

```c
#include <stdio.h>
#include <stdbool.h>

extern bool blackMagic;

int main(int argc, char *argv[]){
    int t1;
    t1 = 8;
    if (blackMagic) {
        t1 = 5;
        printf("t1 is %d\n", t1);
    } else {
        printf("t1 is %d\n", t1);
    }
    return 0;
}
```

The LLVM IR for the above code without any optimization pass:

```
; Function Attrs: noinline nounwind uwtable
define i32 @main(i32, i8**) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  %6 = alloca i32, align 4
  store i32 0, i32* %3, align 4
  store i32 %0, i32* %4, align 4
  store i8** %1, i8*** %5, align 8
  store i32 8, i32* %6, align 4
  %7 = load i8, i8* @blackMagic, align 1
  %8 = trunc i8 %7 to i1
  br i1 %8, label %9, label %12
```

```
; <label>:9:                                        ; preds = %2
  store i32 5, i32* %6, align 4
  %10 = load i32, i32* %6, align 4                   # <- example
  %11 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @.str,
  br label %15

; <label>:12:                                       ; preds = %2
  %13 = load i32, i32* %6, align 4
  %14 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @.str,
  br label %15

; <label>:15:                                       ; preds = %12, %9
  ret i32 0
}
```

Consider the instruction `%10 = load i32, i32* %6, align 4`. Its local dependence would be `store i32 5, i32* %6, align 4`, which is a Def type dependence. Its nonlocal dependence would also includes `%6 = alloca i32, align 4` and `store i32 8, i32* %6, align 4`.

Does the algorithm take advantage of the SSA form? Yes, it does. The SSA form greatly simplfies the memory dependence analysis. During the analysis, we need to get the memory location that a specific instruction accesses. The SSA form reduces the number of possible memory locations that an instruction may accesses. Consider the following instruction:

```
1: int *b;
2: int *c;
... black magic ...
705: int a = *b + *c;
```

Without the SSA form, the instruction at line 705 can access all possible memory locations that are accessed by either b or c. With the SSA form, the instruction at line 705 will be splitted into two loads. We can consider two loads separately, which is arguable simplier and faster to compute.

What transformation in LLVM code can take advantage of -memdep? Interestly, a few AMDGPU targeting passes need memory dependence information:

- /lib/Target/AMDGPU/AMDGPUAnnotateUniformValues.cpp
- /lib/Target/AMDGPU/AMDGPURewriteOutArguments.cpp

The code generation backend also needs the memory dependence information:

- /lib/CodeGen/MachineFunctionPass.cpp

There is one analysis pass that uses the memory dependence information. The analysis prints the memory dependence information to stdout:

- /lib/Analysis/MemDepPrinter.cpp

A handful of transformation passes also use the memory dependence information:

- /lib/Transforms/Scalar/GVN.cpp (global value numbering)
- /lib/Transforms/Scalar/GVNHoist.cpp
- /lib/Transforms/Scalar/MemCpyOptimizer.cpp (memory copy optimization)
- /lib/Transforms/Scalar/DeadStoreElimination.cpp (dead store elimination)