GROUP 13
CS3020
5 November 2015

LLVM Analyses/Transform Passes and Optimizations

Mini-Programming Assignment #3

1. A Brief Introduction To Passes

In LLVM optimisations are implemented as passes. These passes have two main features. They traverse the program to either gather different forms of information regarding the program (Analysis), or they "mutate" the program into a more optimal state (Transform). We will be observing several examples in the report, covering both Analyses and Transform Passes.

2. The Analyses Passes

As mentioned before, the analyses passes gather information which can later on be used for the Transform phase.

Each producing a result that can be queried later. The use of results obtained by Analyses phase is not limited to just Transform passes, other Analyses passes too can use the generated results to form their own. Analyses passes in general form a dependency graph.

One of the most important aspects of Analyses passes is that it does not mutate any of the code, unlike transform passes.

The following are some examples of analyses passes:

2.1. Control Flow Graph:

It is a representation of all the paths that can be traversed during the lifetime of the program once it's executed.

2.1.1. -dot-cfg:

The CFG is printed to a dot file. The creation of this can only be done through opt.

2.1.2. -dot-cfg-only:

prints the control flow graph and omits the body of the graph.

For our sample code, two dot files were generated. One for the main function and the other for the user defined functions. The main function stored the number, made i/o calls and executed the function call in one of the function. Name mangling for the function was observed.

In the factorial program, the recursive nature of the function wasn't so clear. The if condition was completed itself into and if-then-else construct.

Both the return types were stored in some register and then with the use of labels were taken to the block where the value of the function was explicitly returned.

2.2. Alias Analysis:

This is also known as pointer alias. It is used to identify whether two pointers point to the same memory location, it detects their dependencies.

The four main types of aliases in consideration are NoAlias (pointers will never point to the same memory), MayAlias(They may point to same location), PartialAlias(The memory locations that the pointers point to may have some form of an overlap), and MustAlias (They must point to same location).

2.2.1. -basicaa:

Basic Alias Analysis: It is the default alias analysis. It checks for facts such as:

2.2.1.1. Indexes into arrays with statically differing subscripts cannot alias.

2.2.1.2. Structs with different fields do not alias.

2.2.1.3. Pointers that point to constant globals, must not alias. They are fixed.

2.2.2. -tbaa:

Type based alias analyses.

2.2.2.1. Memory has no types.

2.2.2.2. Separate mechanism from policy.

2.2.3. -aa-eval:

Exhaustive Alias Analysis Precision Evaluator: For each function in the program, it simply queries to see how the alias analysis implementation answers alias queries between each pair of pointers in the function.

2.2.4. -count-aa:

Count Alias Analysis Query Responses: A pass which can be used to count how many alias queries are being made and how the alias analysis implementation being used responds.

The getMustAliases method returns all values that are known to always must alias a pointer.

The pointsToConstantMemory method returns true if and only if the analysis can prove that the pointer only points to unchanging memory locations.

2.3. Dominator Tree:

Node m of a CFG dominates node n if every path from the entry node of the graph to n passes through m.

2.3.1. *-domtree:* A simple dominator construction algorithm for finding forward dominators.

2.3.2. *-dot-dom-only:* Prints dominance tree of function to “dot” file without function bodies.

2.3.3. *-dot-dom:* Print dominance tree of function to “dot” file.

2.3.4. *-dot-postdom: P*rints post dominance tree of function to “dot” file

2.3.5. *-dot-postdom-only:* Print post dominance tree of function to “dot” file without function bodies.

Like the CFG class, the Dominator passes create 2 separate dot files, one for the main function and the other for the factorial function. Unlike the other graphs however, the graphs manifested from the Dominator passes do not represent a flow. In the main file there are 4 blocks. The first set of lines that represent the declaration of variables and initiating their values by taking user input are the parent block. The other 3 blocks, namely the branches of the if condition and the return value are all shown to derive from it.

Even in the other graph that represents the factorial function, the hierarchy is represented in a similar fashion as that of the main function. Here too, the initialising block is the parent and the subsequent if-then-else branches and return block are its children.

2.4. Call Graph:

A call graph is a directed graph that represents calling relationships between subroutines in a computer program.

Each node represents a procedure and each edge (f, g) indicates that procedure f calls procedure g.

Thus, a cycle in the graph indicates recursive procedure calls.

2.4.1. *-dot-callgraph:* Prints the call graph to a .dot file

2.4.2. *-print-callgraph:* Prints to standard error.

2.4.3.*-print-callgraph-sccs:* Print the SCCs of Call Graph. Since the cycles are evident in the first two representations, a sorted (for example topological sort), is required to get a clear picture of all the calls involved.

2.4.3.1.Example:

```
      SCCs for the program in PostOrder:
SCC #1 : external node,
SCC #2 : printf,
SCC #3 : scanf,
SCC #4 : _Z9factoriali,  (Has self-loop).
SCC #5 : main,
SCC #6 : external node,
```

Options 2 and 3 printed the output to the terminal. The output of the first and the second command were similar. The graphical representation of the

callgraph is inserted. It shows that apart from the calls made by the scan and printf (Node0x7f…) all the other calls are made through an external node. The main function calls upon scan and printf along with the factorial function (name mangling is evident). Recursive properties of the factorial function are also highlighted.

3. The Transform Passes

Unlike Analyses Passes, Transform passes modify the application code.

While using transformation passes we have to observe the following:
- All passes must leave the IR in a valid state.
- Can depend on analysis pass results.
- Can preserve analysis pass results even while transforming IR.

3.1.Dead Code Elimination:

3.1.1.-die: Dead Instruction Elimination:

It performs a single pass over the function, removing instructions that are obviously dead.

An example is removal of an unused variable that I inserted in the program.

3.1.2.-dce: Dead Code Elimination:

Unlike die, dce implements multiple passes to check for newly created dead instructions.

3.1.3.-deadargelim: Dead Argument Elimination:

This removes arguments which are directly dead and the arguments only passed into function calls as dead arguments of other functions.

Additional parameters in the factorial function that were not used were removed by this command. I had defined an int fake and passed it without usage.

3.1.4.-adce: Aggressive Dead Code Elimination:

This works the same as -dce but it assumes that values are dead until proven otherwise.

3.2. -constprop: Constant Propogation:

This pass looks for instructions that involve only constants. If such an instruction is found it replaces it with the constant itself.

Currently there were no constants in the recursive program that we have implemented. However, an example can be seen as follows:

```
sub i32 9, 1
```

will be replaced by

```
i32 8
```

The following two are loop based code optimisations. A different code was used to check for the alterations in the generated code.

3.3.-licm: Loop Invariant Code Motion

In test2.cpp a constant variable is initilized in the loop over and over. x = 5. The corresponding llvm code was:

```
; <label>:5

; preds = %2

    store i32 5, i32* %x, align 4
```

After the loop invariant pass, this constant definition was taken out of the loop.

```
store i32 5, i32* %x, align 4
```

This was found in the main block and not in the block for the loop.

3.4.-loop-unroll:Unroll loops

4. The Source Code

The source code for the passes can be found in the following directory:

/usr/local/Cellar/llvm/3.6.2/include/llvm

COMMON

The segments of code that are common to both the Analyses and Transform passes are stored in this directory. These include: InitializePasses, AnalysisSupport, PassInfo, PassManager, PassRegistry, and PassSupport header files.

- **Initialize passes:** This file contains the declarations for the pass initialization routine for the entire LLVM project. It has a datatype PassRegistry which is declared as a class member. Each pass for the entire LLVM is initialised in a uniform manner by passing its corresponding PassRegistry as an initialising parameter. For example:

  void initializeCFGSimplifyPassPass(PassRegistry&);

- **LinkAllPasses:** It pulls in all the passes from the Analyses and Transform passes to form the given tools. Like Initialize Passes, its creation manner is uniform for all passes with it calls the create method from each pass Class and returns a void type. The link is forced by creating a global definition.

  (void) llvm::createCFGSimplificationPass();

- **Pass.h:** This is the base class for all Transform passes, all passes (transform) should extend from one of the passes specified in this file. There are mainly 7 types of passes that can be defined, including: basicBlock, Region, Loop, Function, CallGraphSCC, Module, and PassManager. Different types of PassManagers are specified, and for each pass a suitable manager is identified. This also includes PassSupport and PassAnalysesSupoort so that the API's used by these headers are also available for Transform Pass.
- **AnalysisPass:** This is the base class for the Analyses passes. However, since the Pass.h header file explicitly calls this. This header file should NOT be called directly in any other pass.
- **PassManager:** A helper header used for new clients who are currently out of tree users.
- **PassRegistry:** This is used to initialise and register passes. The entries in the Register entry are later on used to resolve dependency resolutions. In order to make the registry thread safe, the use of mutex is incorporated.

ANALYSES:

- **Passes:** This file defines prototypes for accessor functions. It is a gateway to the passes in the analysis libraries. There are 7 classes of passes that can be formulated in the Pass. These include FunctionPass, ImmutablePass, LoopPass, ModulePass, Pass, PassInfo, and LibCallInfo. Each pass returns on of these 7 classes when created in the methods of the class.
- **AliasAnalysis:** This file works with pointers to memory and checks for aliases. i.e to see if two pointers point to the same memory location. Pointers that are equal but point to constant memory are not aliases since they are not dependent on one another.
    - There are 4 main aliases that are stored in an enum known as AliasResult. These aliases include: NoAlias, MayAlias, PartialAlias, and MustAlias.
    - All passes must get initialised through the InitailizeAliasAnalysis method.
    - Alias queries are checked using a Location structure. This structure notes the starting memory of the pointer, the size of he location allocated, and the metadata nodes that describe the aliasing.
    - Even the results of the Aliasing check (the returns from mod/ref behaviour) are identified using datatypes stored in enums. Wrapper functions are used extensively during this stage.
- **CFG**: This performs analyses on the basic blocks and the instructions contained in them. It uses various attributes and methods of the basicBlock class as well as the CFG class from the LLVM IR. There are 4 main checks that it performs on the basic block, including:
    - **FindFunctionBackends:** Finds all the loop back edges of a given function, returning their result in edge form.
    - **GetSuccessorNumber:** For blocks that have a successor BasicBlock *Succ it calls the successor of the given block BasicBlock *BB.
    - **isCriticalEdge:** Checks if the edge is from a block with multiple successors to a block with multiple predecessors.
    - **isPotentiallyReachable:** This walks down through successors to see if the block *to is reachable from the the block *from.
- **DominanceFrontier.h:** This is used in the creation of the dominator tree. The main definitions are implemented in the Dominators code, present in LLVM IR. This current version is non-extensible for further use.

TRANSFORM:

- **Vectorization.h:** It defines prototypes that allow passes to be used by Vectorize transformation library. It uses basic blocks and their passes and vectorises them. It first configures the Vectorization process from the command line options and then created vectorised instances of aspects such as basic blocks and loop passes.
- **UnrollLoop:** Unrolling transformations happen using one main function. This function is named UnrollLoop and has a boolean return type. Apart from passing the actual Loop and the LoopInfo, all transformation on passes must go through the Pass Manager as mentioned earlier. In this case, the LPPassManager.

6. Auto Vectorization

LLVM Support vector type and its operation in IR level. It generates vector type to MMX/SSE instruction in IA32 architecture.

LLVM consists of two types of vectorizers, Loop Vectorizers and SLP vectorizers.

Vectorization does the following things on higher level:

1- Find a loop

2- Check if it is vectorizable?

3- If so, vectorize it.

Vectorization example:

```
int a[259], b[259], c[259];

for(i=0;i<259;++i) {

 a[i] = b[i+1] + c[i];

}
```

This loop is vectorized to,

```
int a[259], b[259], c[259];

for(i=0;i<259; i+=8) {

 a[i:i+8] = b[i+1:i+9] + c[i:i+8];

}

if (i!=259) {

for(;i<259;++i) {

a[i] = b[i+1] + c[i];

}

}
```

The vectorization optimized the IR code and consisted 3 main sections,

Vectorized loop body:

Inserted bitcast instruction after every getelementptr insturction.

Replaced uses of getelementptr to use bitcast If it is a Load or Store instruction, set alignment constraint.

Construct set of instructions which requires type casting from array/pointer type to vector type.

Maximal use set of getelementptr and Type cast instructions in type casting set to vector type.

Modify increment of induction variable to vectorization factor 5. Modify destination of loop exit to epilogue preheader

2. Epiogue preheader:

Generate epilogue preheader.

If there are remainders, jump to epilogue loop.

3. Epilogue Loop:

Generate epilogue loop for remainder.

Clone original loop body.

Update all the uses to denote the cloned one.

Update phi of induction variable.

7. POLLY

It is always a tedious task to manually analyze and detect parallelism in programs. When we deal with auto-parallelism the task becomes more complex.

Frameworks such as OpenMP is available through which we can manually annotate the code to realize parallelism and take the advantage of underly multi-core architecture. But the programmer's life becomes simple when this is done automatically. Polyhedral optimizations in LLVM(POLLY) was birthed on that premise.

Polly is designed as a set of compiler internal analysis and optimization passes.

They can be divided into front end, middle end and back end passes. The front end translates from LLVM-IR into a polyhedral representation, the middle end transforms and optimizes this representation and the back end translates it back to LLVM-IR. To optimize a program manually three steps are performed. First of all the program is translated to LLVM-IR. Afterwards Polly is called to optimize LLVM-IR and target code is generated. The LLVM-IR representation of a program can be obtained from language-specific LLVM based compilers(Usually clang, although Polly also provides a gcc like user interface called pollycc).

REFERENCES

http://llvm.org/docs/Passes.html

http://llvm.org/docs/WritingAnLLVMPass.html

http://llvm.org/docs/Vectorizers.html

http://www.cs.ucla.edu/~pouchet/software/polybench/