Understanding Changes made by a Pass in the Opt Pipeline

—

Jamie Schmeiser

IBM

Agenda Part I

Agenda Part II

Background: Compiler Phases

Front End (clang)

- Preprocessor
- Parser
- Semantic checking
- Intermediate Representation (IR) Generation

Back End (llvm)

- Optimization
- Code Generation
  - Typically some optimization here
- Linking
  - May be optimization here also

Background:
Optimization in Clang C/C++

Optimization is optional

Default is no optimization

Several different levels, specified by options

- -O, -O1, -O2 –O3
- Each level of opt may do
  - Different optimizations
  - Different orders
  - Optimize in different phases

Here we are only concerned with optimization in llvm before code generation.

Background: LLVM

There are multiple meanings

- Here, we typically mean the LLVM IR
  - A machine independent representation of some or all of a semantically checked program
  - Exists in binary form but typically examined in text form
  - Refer to LLVM Language Reference Manual
    - https://llvm.org/docs/LangRef.html

Background: Passes

Work on LLVM IR

Valid LLVM in / Valid LLVM out

Act on different amounts of code

- Module
- Function
- Loop
- Etc

May or may not change IR

Typically do a single or related set of transformations

Analysis Passes provide information about IR

- Do not change IR
- Used by other passes

Background:
Opt Pipeline

Numerous passes which work in sequence

Passes are called by Pass Manager

– Also calls the analysis passes

Portion of LLVM sent through pipeline

– First pass operates on piece of code, then next pass operates on same piece, etc
  - Eg, function goes through pipeline, then next function is sent through

Understanding optimization is understanding how the passes change the IR as it flows through the pipeline

– Need to know what changes each pass makes

– Need to know order that changes are made

Background: Pass Managers

There are 2 pass managers

- Different calling conventions for passes
  - Most passes support both
  - Some new passes may not support Legacy Pass Manager
- Legacy Pass Manager
  - Current default
  - Being phased out
- New Pass Manager
  - Future direction
  - Accessed from clang/clang++ using –fexperimental-new-pass-manager

Background:
Opt Executable

A utility program that allows a custom pipeline to be built and tested

Desired passes are listed in options

- Some passes will be added
  - Eg, Verification passes
- LLVM as input / LLVM as output
- Uses Legacy or New Pass Manager based on options
- Used for unit testing
- <build-directory>/bin/opt

Supports options that show IR on command line

- Eg, opt –S test.ll –print-before-all –O2

These options can also be used with clang/clang++ by prefixing them with –mllvm

- Eg, clang test.c –O2 –mllvm –print-before-all
- If only interested in option output
  - Use –disable-output
  - Or redirect stderr to stdout, direct stdout to /dev/null and pipe stdout
    - Eg, opt test.ll –print-before-all –O2 2>&1 > /dev/null | more

Motivation:
Why look at Opt Pipeline?

Debugging

– The pipeline is producing bad code or crashing

– A performance regression needs to be investigated

New Development

– Is the pass doing what is expected

– What are the partial results/changes

Learning

– What is a pass doing?

– How do the passes interact

Opt exe Options: Sample C Program

```
int summation(int N) {
  int I;
  int Total = 0;
  for (I = 1; I <= N; ++I) {
    Total += I;
  }
  return Total;
}
int main(int argc, char **argv) {
  return summation(5);
}
```

To get the initial llvm ir for test.c to pass into opt:

clang test.c -emit-llvm -c -S -O2 -Xclang -disable-llvm-passes -o test.ll

– Gotcha:

- If you do not specify –O2, then the IR will indicate that it is not to be optimized and opt will not optimize the input.

Opt exe Options:
Determining What the Opt Pipeline is Doing

Traditional ways of determining what the compiler is doing when optimizing a program:

- Debug information
- Optimization Statistics
- Options to opt pipeline
  - Prefix with –mllvm for clang/clang++
- Special passes in pipeline
- Optimization Remarks Emitter

Opt exe Options: -debug-only=<debug id> and -debug

Eg: opt –S test.ll –debug-only=instcombine –O2 2>&1 > /dev/null

Eg: clang test.c –mllvm –debug-only=instcombine –O2

Each file may have a macro DEBUG_TYPE

- Use the macro name as <debug id>
- Use –debug to get debug output from all passes
- Runs code in LLVM_DEBUG macros in file
- Multiple files may use same DEBUG_TYPE macro
- Need to examine file to find macro names to use and the meaning of the output

- Intended for debugging individual passes
- Not intended for general consumption
- The output may be specific to certain aspects of the pass
- You can add more code in LLVM_DEBUG macros
  - Code is typically “dbgs() << …” but it can be any code, including multiple statements
  - Be careful that such code does not accidently introduce side-effects
  - Value::dump() can be used
  - Clang-format may do weird formatting in these macros…

Opt exe Options: debug-only=instcombine output

```
INSTCOMBINE ITERATION #1 on summation
IC: ADD:   br label %for.cond
IC: ADD:   %inc = add nsw i32 %I.0, 1
IC: ADD:   %add = add nsw i32 %Total.0, %I.0
IC: ADD:   ret i32 %Total.0
IC: ADD:   br i1 %cmp, label %for.body, label %for.end
IC: ADD:   %cmp = icmp sle i32 %I.0, %N
IC: ADD:   %Total.0 = phi i32 [ 0, %entry ], [ %add, %for.body ]
IC: ADD:   %I.0 = phi i32 [ 1, %entry ], [ %inc, %for.body ]
IC: ADD:   br label %for.cond
           ...
IC: Visiting:   %I.0 = phi i32 [ 1, %entry ], [ %inc, %for.body ]
IC: Visiting:   br label %for.cond


INSTCOMBINE ITERATION #2 on summation
IC: ADD:   ret i32 %Total.0
IC: ADD:   br label %for.cond
IC: ADD:   %inc = add nuw nsw i32 %I.0, 1
IC: ADD:   %add = add nuw nsw i32 %Total.0, %I.0
IC: ADD:   br i1 %cmp.not, label %for.end, label %for.body
IC: ADD:   %cmp.not = icmp sgt i32 %I.0, %N
IC: ADD:   %Total.0 = phi i32 [ 0, %entry ], [ %add, %for.body ]
           ...
```

Opt exe Options: -stats

Eg: opt –S test.ll –stats –O2 2>&1 > /dev/null

Eg: clang test.c -mllvm -stats -O2

- Prints out statistics at end of compile
- Supported in both pass managers
- Ad hoc method
- STATISTIC(<id>, <string>) declares <id> and registers it.
- If –stats specified and <id> is non-zero, it is reported at end of compile as

  <id value> <name> - <string>

- There can be multiple STATISTIC macros in a file
- One can add new ones to help understand aspects of the compile
- Grepping for a string in the statistics output can be used to quickly determine whether a transformation succeeded when developing code.

Opt exe Options:
-stats output

```
===-------------------------------------------------------------------------===
                          ... Statistics Collected ...
===-------------------------------------------------------------------------===

 1 cgscc-passmgr            - Maximum CGSCCPassMgr iterations on one SCC
 2 correlated-value-propagation - Number of no-signed-wrap deductions for add
 2 correlated-value-propagation - Number of no-wrap deductions for add
 2 correlated-value-propagation - Number of no-signed-wrap deductions
 1 correlated-value-propagation - Number of no-unsigned-wrap deductions
 3 correlated-value-propagation - Number of no-wrap deductions
 1 correlated-value-propagation - Number of no-unsigned-wrap deductions for shl
 1 correlated-value-propagation - Number of no-wrap deductions for shl
 1 function-attrs           - Number of arguments marked nocapture
 2 function-attrs           - Number of functions marked as norecurse
 2 function-attrs           - Number of functions marked readnone
 1 function-attrs           - Number of arguments marked readnone
 2 globalopt                - Number of globals deleted
 2 globalopt                - Number of globals marked unnamed_addr
 4 globalsmodref-aa         - Number of functions that do not access memory
 4 globalsmodref-aa         - Number of functions that only read memory
 1 gvn                      - Number of blocks merged
 1 indvars                  - Number of exit values replaced
 1 inline                   - Number of functions inlined
 1 inline-cost              - Number of call sites analyzed
 6 instcombine              - Number of insts combined
 2 lcssa                    - Number of live out of a loop variables
 1 loop-delete              - Number of loops deleted
 1 loop-rotate              - Number of loops rotated
 1 loop-unswitch            - Total number of instructions analyzed
 2 mem2reg                  - Number of PHI nodes inserted
 4 mem2reg                  - Number of alloca's promoted with a single store
 1 reassociate              - Number of insts reassociated
 2 scalar-evolution         - Number of loops with predictable loop counts
 3 simplifycfg              - Number of blocks simplified
 1 sroa                     - Maximum number of partitions per alloca
 7 sroa                     - Maximum number of uses of a partition
18 sroa                     - Number of alloca partition uses rewritten
 6 sroa                     - Number of alloca partitions formed
 6 sroa                     - Number of allocas analyzed for replacement
22 sroa                     - Number of instructions deleted
 6 sroa                     - Number of allocas promoted to SSA values
```

Opt exe Options:
-print-[before | after]-all

Eg: opt -S test.ll -print-before-all -O2 2>&1 > /dev/null

Eg: clang test.c -mllvm -print-after-all -O2 2>&1 > /dev/null

- Prints the IR before and after each pass is called, respectively
- Supported in both pass managers
- Prints the IR that the pass handles
  - Eg: Function pass will print function IR
  - Prints banner “*** IR Dump …” before dumping IR
- Inconsistent banners between pass managers
- clang test.c -mllvm -print-after-all -fexperimental-new-pass-manager -O2 2>&1 > /dev/null | grep "*** IR" | wc
  - Counts number of passes run
- Also reports Code gen passes with banner prefixed with ‘#’
- clang test.c -mllvm -print-after-all -fexperimental-new-pass-manager -O2 2>&1 > /dev/null | grep "*** IR" | sed “/# *** IR/d” | wc
  - Counts number of opt passes run

Opt exe Options:
print-[before | after]-all continued:

- Useful for discovering IR before or after a pass
- Can be used together to get IR both before and after a pass to determine what a pass did
- Options have Problems:
  - Volume of output
    - Large compile could easily have thousands of passes run
  - Prints actual IR, which is not typically enough to use as input to opt
    - Missing supporting declarations

Two supporting options help address these problems

- Supporting options have no meaning in isolation

Opt exe Options:
print-after-all output with new pass manager

```
...
*** IR Dump After PromotePass ***
   ...
*** IR Dump After PromotePass ***
   ...
*** IR Dump After DeadArgumentEliminationPass ***
; ModuleID = 'test.c'
source_filename = "test.c"
target datalayout = "e-m:e-i64:64-n32:64"
target triple = "powerpc64le-unknown-linux-gnu"

; Function Attrs: nounwind
define dso_local signext i32 @summation(i32 signext %N) local_unnamed_addr #0 {
                ...
for.cond:                                         ; preds = %for.body, %entry
  %I.0 = phi i32 [ 1, %entry ], [ %inc, %for.body ]
  %Total.0 = phi i32 [ 0, %entry ], [ %add, %for.body ]
  %cmp = icmp sle i32 %I.0, %N
  br i1 %cmp, label %for.body, label %for.end

  ...
}
; Function Attrs: nounwind
define dso_local signext i32 @main(i32 signext %argc, i8** %argv) local_unnamed_addr
#0 {
      ...
}
```

```
...
*** IR Dump After InstCombinePass ***
; Function Attrs: nounwind
define dso_local signext i32 @summation(i32 signext %N) local_unnamed_addr #0 {
                ...
for.cond:                                         ; preds = %for.body, %entry
  %I.0 = phi i32 [ 1, %entry ], [ %inc, %for.body ]
  %Total.0 = phi i32 [ 0, %entry ], [ %add, %for.body ]
  %cmp.not = icmp sgt i32 %I.0, %N
  br i1 %cmp.not, label %for.end, label %for.body

                ...
}
*** IR Dump After SimplifyCFGPass ***
   ...
```

Opt exe Options: -filter-print-funcs

Eg: clang test.c -mllvm -print-after-all -fexperimental-new-pass-manager -O2 -mllvm -filter-print-funcs="summation“

- Option takes a comma separated list of functions names
- Only reports passes operating on that function
- Supported in both pass managers
- Supports both print-before-all and print-after-all
- Note: For C++, need to specify the mangled name
  - Eg: -filter-print-funcs=“_Z9summationi”
- Need to look at IR when multiple functions are filtered to determine what function IR is for
  - Banner does not have name of function
- Need to use both –print-before-all and –print-after-all or find previous IR for function to determine changes to IR

Opt exe Options: -print-module-scope

Eg: clang test.c -mllvm -print-before-all -fexperimental-new-pass-manager -O2 -mllvm -print-module-scope

– Instead of reporting the IR that the pass receives, it prints the IR for the module
  - Useful for getting an IR that can be fed into the opt pipeline
  - May increase size of output dramatically

The two options can be combined to limit the output

– Use –filter-print-funcs to limit output to a single function and combine it with –print-module-scope to get an IR that can be fed into the opt exe

– Then specify a single pass in the options to opt to determine the changes the pass makes to the IR

Opt exe Options: -print-before, -print-after

Eg: opt -S test.ll -print-before=instcombine -O2

- Takes a list of passes
- Prints IR before and after, respectively, the indicated passes
- Does not work with clang/clang++ as option handling complains
- Does not work as expected with new pass manager
  - work is being done to get it to work
- Can be combined with –filter-print-funcs
  - IR from specified pass for specified function

Opt exe Options: Combining Techniques

Combine –stats, –print-before, -print-after

- clang test.c -mllvm -stats -O2
  - 6 instcombine - Number of insts combined
- find src/llvm –name "*.cpp" | xargs grep "Number of insts combined"
  - src/llvm/lib/Transforms/InstCombine/InstructionCombining.cpp
- opt -S test.ll -O2 -print-before=instcombine -print-after=instcombine -filter-print-funcs=summation 2>&1 > /dev/null | grep "*** IR"

```
*** IR Dump Before Combine redundant instructions ***
*** IR Dump After Combine redundant instructions ***
*** IR Dump Before Combine redundant instructions ***
*** IR Dump After Combine redundant instructions ***
*** IR Dump Before Combine redundant instructions ***
*** IR Dump After Combine redundant instructions ***
*** IR Dump Before Combine redundant instructions ***
*** IR Dump After Combine redundant instructions ***
*** IR Dump Before Combine redundant instructions ***
*** IR Dump After Combine redundant instructions ***
*** IR Dump Before Combine redundant instructions ***
*** IR Dump After Combine redundant instructions ***
*** IR Dump Before Combine redundant instructions ***
*** IR Dump After Combine redundant instructions ***
*** IR Dump Before Combine redundant instructions ***
*** IR Dump After Combine redundant instructions ***
```

Special passes: dot-cfg and dot-cfg-only

Eg: opt -S test.ll -passes=dot-cfg > /dev/null 2>&1

Eg: dot -Tpdf -o summation.pdf .summation.dot

– The dot-cfg analysis pass creates a dot file for each function in IR
  - File named .<function-name>.dot
  - Use ls –a to see hidden files (that start with .)
  - Dot produces pdf (as specified above) with basic blocks forming the control flow graph (CFG) with instructions shown

-dot-cfg-only just shows CFG without the instructions

Useful for spotting IR patterns visually

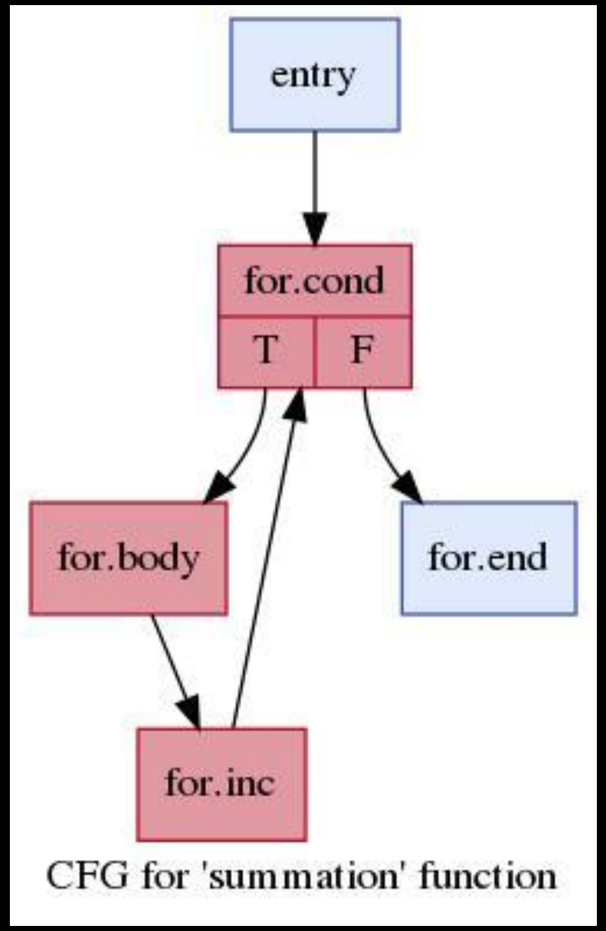
– Eg, Loops are easier to understand

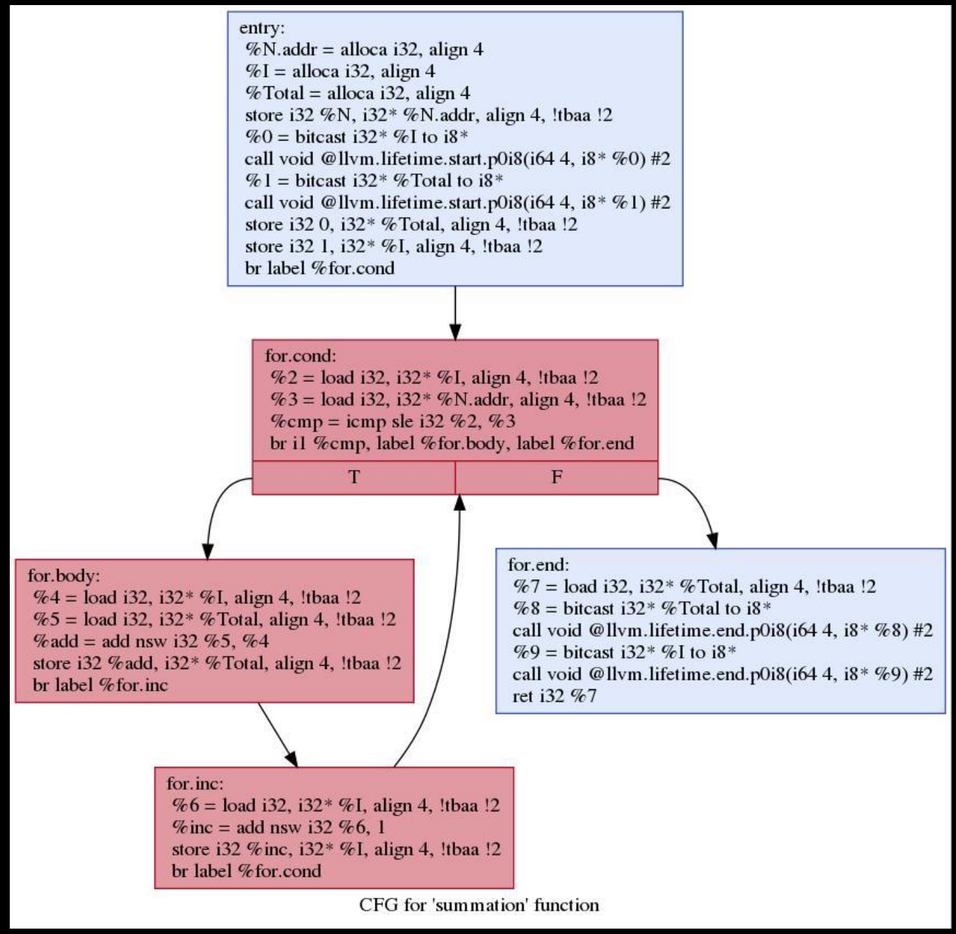
Dot can produce different formats

– PDF is useful for scaling

– Used jpeg for presentation

Special passes:
dot-cfg-only and dot-cfg Examples





Summary of Existing Ways: Doable but Inconvenient

At present, these are ways to determine what a pass is doing:

- Use debug information
- Use statistics
- Print before or after passes with some filtering
- Generate a dot file by adding pass into the pipeline

Difficulties:

- Debug information is based on what the developers wanted to debug the code
- Statistics gives some idea of what was done but not by which pass.
- Printing before/after requires inspection of output with no indication of what passes actually changed the IR. Filtering can hide passes that made changes.
- Dot files requires capturing the IR using the previous printing methods and creating an artificial pipeline then running dot.

Part II
New Ways of Determining What is Happening

New Options:
New Ways of Examining Passes:

There are several new ways of examining the IR

- Code under review but should land soon
- Subject to change but concepts should remain
- Code is designed to be extendable to allow one to easily add new variations
- They build upon each other becoming more useful and convenient as they progress so they will be presented in order
- Each has particular uses and benefits

Note that these new options are only available with the new pass manager

- Requires –fexperimental-new-pass-manager for clang/clang++
- Need to construct pipeline that specifies new pass manager when using opt

New Options: -print-changed

A major difficulty with –print-[before | after]-all is that it does not indicate whether a pass changed the IR

Need to edit output, extract IRs and do diffs until changes are found

-print-changed filters out all passes that do not change the IR

- Prints initial IR of module
- Prints the IR after each pass that changes the IR
- Those passes that do not change the IR just have the banner indicating what they were and why they were not reported
- Automatically filters out some Pass Manager grouping passes and invalidated passes
- Only in new Pass Manager
- Repects –filter-print-funcs
- Respects –print-module-scope
- New supporting option –filter-passes=<list of passes>
  - Takes list of passes
  - Names are the PassIds used in reporting
- These options can be combined

New Options:
-print-changed Example

Eg: clang test.c -fexperimental-new-pass-manager -O2 -mllvm -print-changed

```
*** IR Dump At Start: ***
; ModuleID = 'test.c’
        ...
*** IR Dump After EntryExitInstrumenterPass (function: main) omitted because no change ***
*** IR Pass ModuleToFunctionPassAdaptor<llvm::EntryExitInstrumenterPass> (module) ignored ***
*** IR Dump After InferFunctionAttrsPass (module) omitted because no change ***
*** IR Dump After SimplifyCFGPass *** (function: summation)
; Function Attrs: nounwind
define dso_local signext i32 @summation(i32 signext %N) #0 {
entry:
  %N.addr = alloca i32, align 4
        ...
```

New Options:
-print-changed Removes 85% of Useless Information

Six types of banners in output

- *** IR Dump At Start: ***
- *** IR Dump After <ID> *** (<NAME>)
- *** IR Pass <ID> (<NAME>) ignored ***
- *** IR Pass <ID> invalidated ***
- *** IR Dump After <ID> (<NAME>) omitted because no change ***
- *** IR Dump After <ID> (<NAME>) filtered out ***
  - <ID> is pass name
  - <NAME> is name of IR

208 banners produced (207 passes plus initial IR)

Filtering out omitted, invalidated and ignored banners leaves 21 banners

- Only 20 of 207 passes changed the IR
- -print-after-all produces 7370 lines of output while –print-changed produces 700
- 85% reduction of output plus the remaining information is useful and understandable
  - Just initial IR, changed IR after each pass and banners indicating what each pass did remains

New Options:
-print-changed Filtered Banners

... | grep "*** IR" | sed "/omitted/d;/invalidated/d;/ignored/d“

```
*** IR Dump At Start: ***
*** IR Dump After SimplifyCFGPass *** (function: summation)
*** IR Dump After SROA *** (function: summation)
*** IR Dump After SROA *** (function: main)
*** IR Dump After GlobalOptPass *** (module)
*** IR Dump After InstCombinePass *** (function: summation)
*** IR Dump After PostOrderFunctionAttrsPass *** (scc: (summation))
*** IR Dump After LCSSAPass *** (function: summation)
*** IR Dump After LoopRotatePass *** (loop: %for.body)
*** IR Dump After SimplifyCFGPass *** (function: summation)
*** IR Dump After InstCombinePass *** (function: summation)
*** IR Dump After LoopSimplifyPass *** (function: summation)
*** IR Dump After LCSSAPass *** (function: summation)
*** IR Dump After IndVarSimplifyPass *** (loop: %for.body)
*** IR Dump After GVN *** (function: summation)
*** IR Dump After CorrelatedValuePropagationPass *** (function: summation)
*** IR Dump After DevirtSCCRepeatedPass<llvm::PassManager<LazyCallGraph::SCC, llvm::CGSCCAnalysisManager, llvm::LazyCallGraph &, llvm::CGSCCUpdateResult &> > *** (scc:
(summation))
*** IR Dump After InlinerPass *** (scc: (main))
*** IR Dump After PostOrderFunctionAttrsPass *** (scc: (main))
*** IR Dump After DevirtSCCRepeatedPass<llvm::PassManager<LazyCallGraph::SCC, llvm::CGSCCAnalysisManager, llvm::LazyCallGraph &, llvm::CGSCCUpdateResult &> > *** (scc: (main))
*** IR Dump After ModuleInlinerWrapperPass *** (module)
```

New Options:
-print-changed with –filter-print-funcs

Filters can be used to focus the output

– Use –filter-print-funcs to limit output to specified functions

  - Recall that you need mangled names for C++

Eg: clang test.c -fexperimental-new-pass-manager -O2 -mllvm -print-changed -mllvm -filter-print-funcs=summation

  - Now all passes pertaining to main will just have a banner saying that they were filtered out
  - Passes that do not change summation will still just be reported as not changing the IR
  - ~87% reduction in output in this example

New Options:
-print-changed with –filter-passes

New hidden option to filter passes

– Use –filter-passes=<list of passes> to limit output to specified passes

  - Use ID from banners
  - Now only those passes in the list will be reported (assuming they make a change to the IR) and rest will just have banner that they were filtered out

Eg: clang test.c -fexperimental-new-pass-manager -O2 -mllvm -print-changed -mllvm -filter-passes=InstCombinePass

  - Only initial IR and instances of InstCombinePass that change IR are printed
  - All other passes (including InstCombinePass when it doesn’t change IR) are filtered out
  - ~92% reduction in output in this case

New Options:
-print-changed with multiple filters

The focus can be further narrowed by using both filters at the same time

Eg, clang test.c -fexperimental-new-pass-manager -O2 -mllvm -print-changed -mllvm -filter-passes=InlinerPass -mllvm -filter-print-funcs=main

– Only pass that actually inlines code into main is shown with rest filtered out

– ~93% reduction in output, most remaining output is banners

New Options:
Output from -print-changed with Multiple Filters

```
        ...
*** IR Dump After DevirtSCCRepeatedPass<llvm::PassManager<LazyCallGraph::SCC,
llvm::CGSCCAnalysisManager, llvm::LazyCallGraph &, llvm::CGSCCUpdateResult &> > (module) filtered out
***
*** IR Dump After InlinerPass *** (scc: (main))
; Function Attrs: nounwind
define dso_local signext i32 @main(i32 signext %argc, i8** %argv) local_unnamed_addr #1 {
entry:
  ret i32 15
}
*** IR Dump After PostOrderFunctionAttrsPass (scc: (main)) filtered out ***
        ...
```

New options: –print-before-changed

A new hidden option that modifies the print-changed behavior

- No effect in isolation
- -print-before-changed
  - Prints IR before each pass that changes IR as well as after pass
  - Respects other modifying options
    - IE, it only reports the IR before passes that are reported and not filtered out

New options:
–print-crashed

This new option prints the IR as it existed upon entering a pass that asserts or crashes

Replaces using –print-before-all when the pipeline crashes

Convenient in that only the last IR is reported

It gives a banner indicating which pass was last entered

Traceback still reported

No need to edit a humungous file to get the IR as was needed with –print-before-all

Nothing reported if no crash

Do not use when not needed as it slows compilation

New options: –print-changes

Does same filtering as –print-changed including obeying modifying options but presents output differently

- Rather than just reporting the new IR, it shows the changes in line in the output, similar to a patch
- Removed IR shown prefixed with ‘-’
- Added IR shown prefixed with ‘+’
- Checks changes to IR in basic blocks of function so if pass changes other aspects of IR, it will not recognize the change
  - Eg, changes to just function attributes will not be recognized
- Essentially does combination of –print-changed –print-before-changed with diff of output for each pass that makes a change

New Options:
-print-changes Caveats

The implementation of –print-changes makes a system call to use the linux diff routine

The call uses line formats which is not POSIX diff

So, this option may not work as expected on systems that do not have a diff utility that supports line formats

That said, it presents a very useful view of how passes are changing the IR as it flows through the pipeline

New Options:
-print-changes Example Output with InstCombinePass

Eg: clang test.c -fexperimental-new-pass-manager -O2 -mllvm -print-changes -mllvm -filter-passes=InstCombinePass

```
        ...
*** IR Dump After SimplifyCFGPass (function: summation) filtered out ***
*** IR Dump After InstCombinePass *** (function: summation)

 entry:
-  %cmp6 = icmp slt i32 1, %N
+  %cmp6 = icmp sgt i32 %N, 1
   br i1 %cmp6, label %for.body, label %for.end

for.body:                                         ; preds = %entry, %for.body
        ...
```

New Options:
-print-changes Example Output with SimplifyCFGPass

Eg: clang test.c -fexperimental-new-pass-manager -O2 -mllvm -print-changes -mllvm -filter-passes=SimplifyCFGPass

```
            ...
*** IR Pass PassManager<llvm::Loop, llvm::LoopAnalysisManager, llvm::LoopStandardAnalysisResults &, llvm::LPMUpdater &> (loop: %for.body) ignored ***
*** IR Pass FunctionToLoopPassAdaptor<llvm::PassManager<llvm::Loop, llvm::LoopAnalysisManager, llvm::LoopStandardAnalysisResults &, llvm::LPMUpdater &> > (function: summation) ignored ***
*** IR Dump After SimplifyCFGPass *** (function: summation)
            ...
-  br i1 %cmp.not, label %for.cond.for.end_crit_edge, label %for.body
+  br i1 %cmp.not, label %for.end, label %for.body

-for.body.lr.ph:                                  ; preds = %entry
-  br label %for.body

-for.cond.for.end_crit_edge:                       ; preds = %for.body
-  %split = phi i32 [ %add, %for.body ]
-  br label %for.end
        ...
```

New Options: -dot-cfg-changes

Combines aspects of –print-changes and the dot-cfg analysis pass

Respects filtering options –filter-print-funcs and filter-passes

Creates a simple web-site with a page containing the banners indicating whether each pass changed the IR or was filtered out for some reason.

Banners for passes that changed IR are links to pages showing the control-flow-graph (cfg) as depicted using the linux dot utility (similar to dot-cfg)

Changes are shown in the IR using colour

- Removed shown in red
- Added shown in green
- Unchanged shown in black

New Options:
-dot-cfg-changes (continued)

The option takes the name of an existing directory where it will build the website

Links are relative so copying the all of the files should work

- Main page of website is named passes.html
- Rest of files are PDFs

Eg, clang test.c -fexperimental-new-pass-manager -O2 -mllvm -dot-cfg-changes=<some path>/tutorial

- Initial IR shown with collapsible box
- Module passes show functions indented using 2 level numbering system

New Options:
-dot-cfg-changes (continued)

Colours can be controlled with options

- Use colour names as specified in appendix J of https://www.graphviz.org/pdf/dotguide.pdf
- -before-color=red (default)
- -after-color=forestgreen (default)
- -common-color=black

Caveats:

- Requires same diff capabilities as –print-changes
- dot is called to produce PDFs for graphs

New Options:
Demo Using firefox on Windows

New Options:
Future Enhancements

I have been working on enhancements that are not yet ready:

- Limiting change shown with –dot-cfg-changes to just identifiers rather than whole line
- Recognizing when basic block has name change
- Using collapsible boxes to reduce main webpage size (eg for modules)

Code is designed to allow new change reporters to be easily added

- Base classes determine when changes have occurred and do filtering
- Only need to supply code to show the changes

Concluding Remarks:

Have reviewed existing techniques and strategies for understanding how IR changes as it flows through the opt pipeline.

– Debugging output

– Statistics

– Options to opt

– Special passes to produce graphical representations.

Have introduced and demonstrated new options to help understand changes to the IR.

– -print-changed

– -print-changes

– -dot-cfg-changes

Thank You for Attending

Stay healthy.