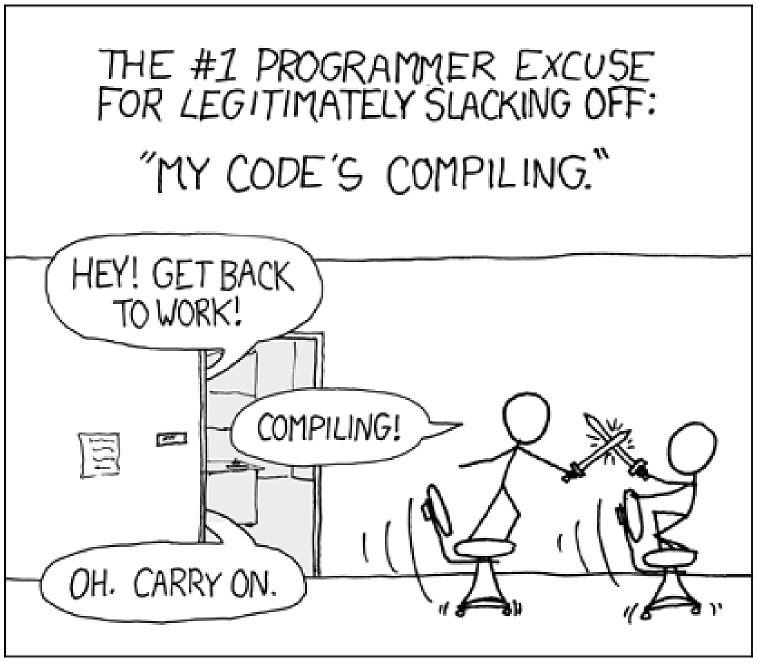


IMAGE CREDIT: `WWW.XKCD.COM`

UNIVERSITY OF TEXAS AT AUSTIN

FINAL PROJECT

# Client-Driven Pointer Analysis in LLVM

*Author:*
Dustin "REACHING DEF" Carlino
Kurt "RAW_STREAM" Scherer
Parth "FLOW INSENSITIVE" Upadhyay
Jason "DEF WITHOUT A USE" Wolfe

*Advisor:*
Dr. Calvin LIN

December 13, 2011

# Contents

## List of Figures

## 1 Introduction

In his 2003 thesis, Samuel Guyer presented the **Broadway Compiler**, a compiler that could take advantage of the *semantics* of library calls to perform optimizations [2]. This compiler was built on the premise that libraries are domain specific languages, and that many existing optimization algorithms could work equally well for library calls. The compiler is typically unable to perform these optimizations without more information about what these library calls are actually doing.

To perform such analyses, Guyer shows that a powerful pointer analysis framework is needed, and he introduces a novel method of client-driven pointer analysis where the specific needs of the client drive the needs of the pointer analysis.

While the ideas behind this project are still valid today, the project has gone largely unmaintained. The goal behind our project is to apply the ideas from Broadway in a modern compiler: LLVM.

We assume familiarity with the general concepts of pointer analysis, context- and flow-sensitivity, and iterative data-flow analysis (IDFA).

## 2 Project Scope and Approach

The specific portion of this task we undertook was porting the client-driven pointer analysis. The original scope of our project included:

- A fast but imprecise flow/context-insensitve pass 1
- A monitor to record loss of precision
- An adapter to use the information from the monitor
- A general IDFA

The scope proved to be too ambitious, however, and we were unable to complete all of the goals we set out to. We were able to complete the general IDFA and the flow/context-insensitive pass as an extension upon the general IDFA with toggle-able sensitivity in either dimension.

Our approach to porting made use of the availability of the original source code. "Clean room reverse-engineering" just by using information from papers may have led us to a cleaner design, but it is unlikely we would have implemented as much of our feature set. There are some significant design details omitted from published papers that are documented in the source. A goal of this report is to make those more clear.

## 3 Program Model

Our program model closely mimics that which was presented in the thesis. However we will explain our understanding and also outline where we deviate from the thesis.

The thesis [2] describes most algorithms in appropriate detail. Section 3.7 describes the changes needed for LLVM. The goal of this section is to discuss the main objects/structures used. We will not repeat explanations of the general algorithms.

We will first give an overview that briefly explains and motivates each structure. Then we will give details on each.

## 3.1 Big Picture

Our aim is for this section to exclusively enable understanding of the algorithms presented in Guyer's thesis, as well as some of those establish by the corresponding code and not necessarily explicitly discussed in the thesis.

The goal is to construct a framework capable of performing context-sensitive (C.S.) and flow-sensitive (F.S.) dataflow analysis. Flow-insensitive (F.I.) and context-insensitive (C.I.) analysis will be degenerate cases of the general engine. Let Memory Blocks ("memblocks", "blocks", "MBs") represent stored memory, whether it's "normal" integer variables on the stack or heap memory. The flow values we will operate on are points-to sets of a MB, meaning one MB can point to a set of MBs.

Traditional iterative data-flow analysis computes and stores flow values at every program point. This is actually quite redundant; it suffices to store the flow value only when it changes. To encode flow-sensitivity, we create a Definition ("def") whenever a MB's flow value changes. At any program point, we can find the reaching def that gives the answer at that point.

What is meant by "program point"? Procedures contain basic blocks, and basic blocks contain statements: so one answer is any of these levels of granularity. However, we also wish to encode context-sensitivity. We want to remember the full call graph from the main() root, rather than the single-level static call-site or any kind of k-limiting. So we can form a tree of Locations: Procedure Locations ("Proc Loc") have Basic Block Locations ("BB Loc") as children, and BB Locs have Statement Locations ("Stmt Loc") as children. The recursive step that gives us C.S. is the optional child of a Stmt Loc: at a procedure call, the Stmt Loc will have a Proc Loc as a child. If the same procedure is called multiple times, it will always have a different Stmt Loc for a parent. This parent is not just the static call-site, but the dynamic call-site. This formulation of Locations is equivalent to cloning procedures.

So we have Memory Blocks with Defs established at Locations. At any Location, we can find the reaching def and get the correct flow value. All MBs are managed by a "Memory Table" that indexes MBs by LLVM SSA value and Proc Loc. When we call a procedure from one location, a local variable may have Defs at various Locations. When we call the procedure from another place, that same local variable should have a different set of Defs. Thus, the Memory Table does need both LLVM value and Proc Loc.

## 3.2 Locations

To encode context-sensitivity, we use the method prescribed in the thesis which involves a tree structure consisting of 3+1 types of nodes:

- Statement Locations (`Stmt_Loc`) - encode single statements. The child is either nothing or a `Proc_Loc`, meaning the `Stmt_Loc` is a call-site.
- Basic Block Locations (`BB_Loc`) - encode basic blocks. The children of a `BB_Loc` are `Stmt_Loc`s that represent the statements contained within the basic block.
- Procedure Locations (`Proc_Loc`) - encode procedures. The children of a `Proc_Loc` are `BB_Loc`s that represent the basic blocks contained with the procedure.
- Global Locations (`Global_Loc`) - encode globals. These aren't really in the tree (no children, no parents), but are used in the same fashion as the rest of the `Loc` instances.

To encode **context sensitivity**, one simply creates a new `Proc_Loc` for every function call. The parent of the Proc Loc encoding the root main() function is null. (It is assumed that no program ever invokes main recursively.) To encode **context insensitivity**, refer to the same `Proc_Loc` for every call to a procedure.

### 3.2.1 Dominance Testing

Finding reaching defs involves making many queries about whether one Location dominates another. Guyer describes an intuitive constant-time numbering scheme for testing the dominates relation. It is difficult to assign the numberings because more Locations are instantiated as procedure calls are analyzed. He mentions an online formulation of this algorithm, but does not detail it. Nothing resembling it appears in the C-Breeze source.

We instead modified the approach used in C-Breeze. Within the same static procedure, answering dominance queries about two statements or basic blocks is trivial – there is even an LLVM pass that does it. Given two Locations of depth $m$ and $n$, we can find their common parent in $O(max(m, n))$ by chasing the deeper Location's parent pointer until the depths are equal, and then chasing both parent pointers simultaneously. The common parent is either a basic block (meaning we have the two Locations as Stmt Locs) or a procedure (meaning we have two BB Locs). At this point, the static LLVM dominator relation can be applied.

In the C.I. case, a Proc Loc's parent will be null. The dominates relation is never true for Locations inside two different C.I. Proc Locs. Finding reaching defs in the presence of C.I. calls is slightly more complicated and may not be implemented correctly.

### 3.2.2 Globals

In LLVM, globals are declared and optionally initialized by a special `global` instruction in the IR. Because they're constant throughout the program, we simply iterate through them before analysis creating `Global_Locs` and `Mem_Defs` for them. Because globals are visible anywhere, they also necessitate a simple change to the dominate test, to the effect that they strictly dominate everything except other globals.

## 3.3 Memory Blocks

A Memory Block (`Mem_Block`) represents anything that can point or be pointed to. This manifests as a normal scalar variable on the stack, a heap-allocated object, a field of a struct, or a return value from a procedure. Our representation stores a list of defs, uses (each of which has a reaching def), and, for structs, a mapping from field to the contained Mem Block.

Our array and heap model follows Guyer's formulation, except for multiple instance analysis of heap variables (described in Limitations). All assignments to an array or heap MB is forced as a weak update, since it may represent multiple objects at run-time.

## 3.4 Externals and Virtual PHI Functions

Virtual phi functions are also introduced whenever a variable external to a function (anything that isn't a purely local variable) is modified inside of a function. Because we place virtual phi functions for merge-points at the start of basic blocks (storing them in the `BB_Loc`), we thus need every function call to result in a unconditional branch to a new basic block.

We take advantage of the modular nature of LLVM by running a simple custom pre-pass `-split-at-call` that takes care of guaranteeing this. By requiring this pass, we have a compiler-infrastructure enforced guarantee that function calls cannot arbitrarily occur in the middle of a basic block.

## 3.5 Def/Use

For `Mem_Block`s we encode def/use information. For defs, we use a `Mem_Def` structure that contains the `Location` of the def and the set of `Mem_Block`s that it points to. For uses, we use a `Mem_Def` structure that contains the `Location` of the use, and the reaching_def of the use.

To adapt our analysis for flow-insensitivity, one must restrict each `Mem_Block` to having only one def which contains the information of all the defs it would have had in a flow-sensitive solution [2].

## 3.6 Procedure DB

C-Breeze pre-computes various things per procedure that do not change. We also adapt this design.

- Guyer references somebody who shows that IDFA converge faster if the all predecessors of a basic block are visited first. We use an LLVM pass to compute the reverse post-order order of basic blocks for a procedure. We adapt their implementation of an always-sorted worklist using bit-sets.
- Mappings of call-sites
- Pre-computed dominance frontiers and ancestor lists

### 3.7 Interaction with LLVM IR

Our rules for evaluating expressions differ slightly from C-Breeze's due to differences in IR.

- When we see an ALLOCA or GLOBAL command, we treat that as an identifier and take an address.
- A LOAD indicates some type of dereference. We do not compute points-to sets for temporary SSA variables; rather, we do all of the work when we hit a STORE. We follow the LOADed operand until we hit an identifier. This tells our dereference mechanism how many times to expand points-to sets.
- LLVM "bitcasts“ always lead to malloc in our tests. In other cases, it should be treated as a no-op.
- GEP (get element pointer) instructions indicate array or struct field accesses.
- CALLs invoke the handling for C.S. or C.I. procedure calls.
- PHI nodes follow the usual semantic of unioning points-to sets. Note that these are not the same as the virtual phis that we insert.
- Constants are no-ops
- Arguments indicate when we should look up the reaching def on a special Mem Block representing a certain parameter.

## 4 Testing

Our original testing plan was to slowly replicate more and more of the original C-Breeze/Broadway results. Since we were unable to ever build it, we instead chose to construct small C programs that test a particular property of our analysis. Some of these programs include:

- passes0: recognizing in LLVM IR basic source operations like taking an address and dereferencing on the left- and right-hand side
- passes1: weak updates caused by a statically unknown conditional
- passes2: having a side effect in a called procedure, so that an interprocedural phi is placed
- passes6: verifying struct fields stay separate and array fields all merge
- testglobal: verifying the behavior of globals
- testci: smearing arguments and return values in C.S. versus C.I. mode

- passes10: testing differences between F.S. and F.I. mode

Each test produces a non-trivial log detailing the analysis, so we used three different techniques to evaluate the validity of the output. We used three techniques to evaluate the results. Each requires manual inspection of the test source just once, to determine the appropriate behavior. This is one reason why we exclusively used small, specific test cases.

With more time, unit tests for many components would have been a wise option. Queries like finding procedure ancestors, detecting recursion, and evaluating location dominance relations would be prime candidates.

## 4.1 Log Comparison

At the end of analysis, the memory table dumps all known Memory Blocks (both regular and synthetic), describing the points-to set and location of each def. Once we verify one log is correct, we could commit and use version control to notice any future changes to that test. This was slightly complicated by the fact that lots of our data structures are unordered sets (resulting in many diffs where results are just printed in a different order). Adding a lexicographic sort on the symbolic names of Memory Blocks alleviates this issue, modulo intermediate variable naming, which is normalized to a standard form by the LLVM InstNamer pass.

## 4.2 Constant Propagation Client

As a test of our callback system for client analyses, we implemented a May analysis copy propagation client. It tracks the set of integer values each non-pointer Memory Block might hold at each location. Information is generated at STOREs involving a constant IR expression. The analysis is incomplete in the presence of function calls. Since we track points-to sets bound to each parameter, we cannot recover from analyzing STOREs the Memory Block a normal data parameter could point to. However, this information could be determined by using a callback for procedure calls. Because this API was in flux until the end of the project, we did not implement this yet.

Constant propagation serves as a summary of the logs described above. A human can statically determine what values a variable might hold at any program point. These results encode less information than all of the points-to sets, since constant values are an effect of the pointer analysis. Less information is faster to manually verify.

## 4.3 Visualization

To visualize the relationships within the model, we harnessed the power of Graphviz, an open source graph visualization software. Graphviz uses a simple text file format that indicates relationships between nodes, and outputs a variety of image formats.

Once the initial hurdles of communication were overcome, we were able to agree on how we wanted the graph to look, and the resulting graphs accurately portray the relationships of the memory model used in our analysis.

When an analysis has been completed, a call is made to dump the current contents of the memory. We the pointers to location objects as node keys in the graph and label the contents of each node accordingly.

Having already built the location relationships, we simply traverse the relationships, outputting as we go. As previously discussed, each location has a clearly defined child-parent relationship, so we only need to call the graph function on the root `Proc_Loc`.

When the graph function is called on the root `Proc_Loc` (typically, this is the `main` function), each child of that `Proc_Loc` is a `BB_Loc`, so it's matter of stepping through each child and outputting the relationship accordingly. Since the nodes are indexed by pointers, it's a reasonably straight-forward task. Following the display of all the `Proc_Loc` to `BB_Loc` relationships, a graph function is called on each `BB_Loc`.

At each `BB_Loc`, the statements within are assigned to the `BB_Loc` in a tabular format, as well as added to a queue. Following the processing of all the statements within a `BB_Loc`, the queue is processed. Each element in the queue is a `Stmt_Loc`, and a graph function is called in it.

The graph function for a `Stmt_Loc` is very simple. If the statement is a call-site, the child will point to a `Proc_Loc`. In this case, the graph function is called on the child `Proc_Loc`. Otherwise, the graphing processes stops there.

Although the original intent of the graph was to incorporate definitions and usages, time constraints limited the success of incorporating the information obtained from the `Mem_Table` into the graph. The chain of graphing functions exist, but we not completed by the deadline.

To illustrate what the current visualization looks like, we'll take the following code block:

```
int* identity (int* in) {
  int *out = in;
  *out = 222;
  return out;
}

int main () {
  int data = 111;
  int data2= 333;
  int *ptr = data == 2 ? &data : &data2;

  *ptr = 222;

  int copy_of_data = *ptr;

  int *clone_ptr;
  clone_ptr = identity(ptr);

  *clone_ptr = 333;
}
```

Here, running the visualization gives us Figure 1.

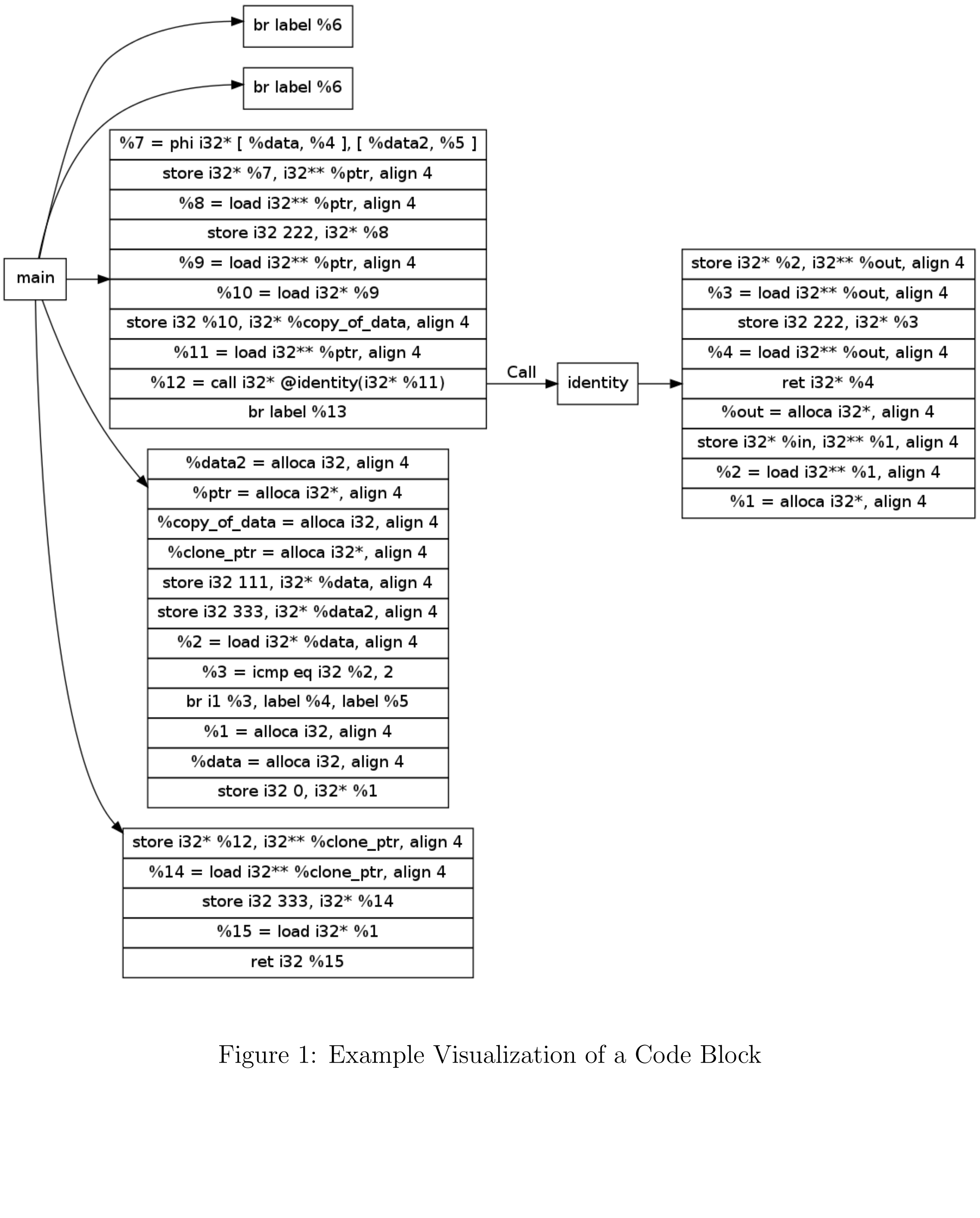
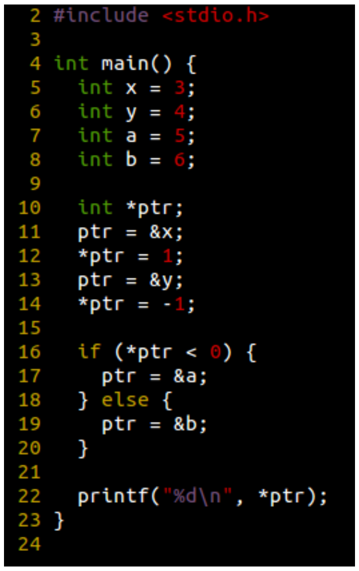
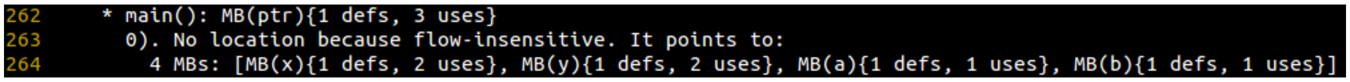

main
br label %6
br label %6
%7 = phi i32* [ %data, %4 ], [ %data2, %5 ]
store i32* %7, i32** %ptr, align 4
%8 = load i32** %ptr, align 4
store i32 222, i32* %8
%9 = load i32** %ptr, align 4
%10 = load i32* %9
store i32 %10, i32* %copy_of_data, align 4
%11 = load i32** %ptr, align 4
%12 = call i32* @identity(i32* %11)
br label %13
Call
identity
store i32* %2, i32** %out, align 4
%3 = load i32** %out, align 4
store i32 222, i32* %3
%4 = load i32** %out, align 4
ret i32* %4
%out = alloca i32*, align 4
store i32* %in, i32** %1, align 4
%2 = load i32** %1, align 4
%1 = alloca i32*, align 4
%data2 = alloca i32, align 4
%ptr = alloca i32*, align 4
%copy_of_data = alloca i32, align 4
%clone_ptr = alloca i32*, align 4
store i32 111, i32* %data, align 4
store i32 333, i32* %data2, align 4
%2 = load i32* %data, align 4
%3 = icmp eq i32 %2, 2
br i1 %3, label %4, label %5
%1 = alloca i32, align 4
%data = alloca i32, align 4
store i32 0, i32* %1
store i32* %12, i32** %clone_ptr, align 4
%14 = load i32** %clone_ptr, align 4
store i32 333, i32* %14
%15 = load i32* %1
ret i32 %15

Figure 1: Example Visualization of a Code Block



```
#include <stdio.h>

int main() {
  int x = 3;
  int y = 4;
  int a = 5;
  int b = 6;

  int *ptr;
  ptr = &x;
  *ptr = 1;
  ptr = &y;
  *ptr = -1;

  if (*ptr < 0) {
    ptr = &a;
  } else {
    ptr = &b;
  }

  printf("%d\n", *ptr);
}
```

Figure 2: Flow-Insensitive Code Example



```
    * main(): MB(ptr){1 defs, 3 uses}
      0). No location because flow-insensitive. It points to:
        4 MBs: [MB(x){1 defs, 2 uses}, MB(y){1 defs, 2 uses}, MB(a){1 defs, 1 uses}, MB(b){1 defs, 1 uses}]
```

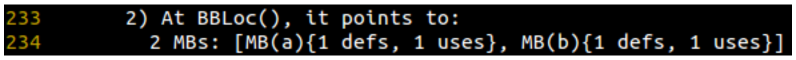
Figure 3: Flow-Insensitive Solution

# 5 Results

## 5.1 Flow-Sensitivity and Flow-Insensitivity

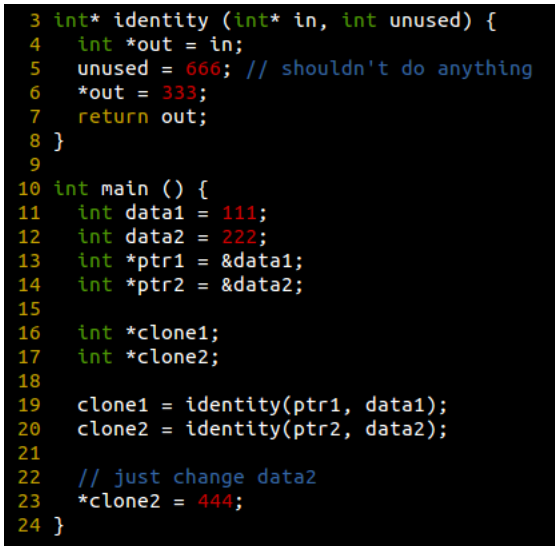
To show that our flow-insensitive and flow sensitive analyses work, consider the following code snippet in Figure 2.

A flow-sensitive analysis should produce that, at the end of the program, `ptr` points only to `a` and `b`. However, a flow-insensitive manuever would show that `ptr` could point to any of `a`, `b`, `c`, and `d`. (Note that we do not use the thesis's version of flow- insensitivity here; we do a regular flow-insensitive solution). In Figures 3 and 4 we see that this is exactly what has happened.



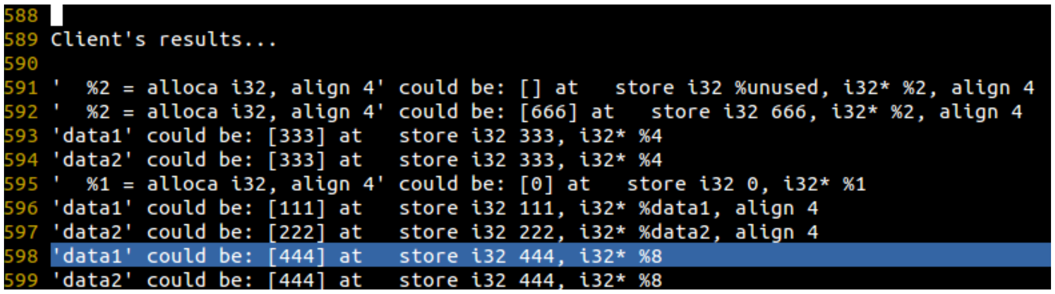
```
      2) At BBLoc(), it points to:
        2 MBs: [MB(a){1 defs, 1 uses}, MB(b){1 defs, 1 uses}]
```

Figure 4: Flow-Sensitive Solution



```
int* identity (int* in, int unused) {
  int *out = in;
  unused = 666; // shouldn't do anything
  *out = 333;
  return out;
}

int main () {
  int data1 = 111;
  int data2 = 222;
  int *ptr1 = &data1;
  int *ptr2 = &data2;

  int *clone1;
  int *clone2;

  clone1 = identity(ptr1, data1);
  clone2 = identity(ptr2, data2);

  // just change data2
  *clone2 = 444;
}
```

Figure 5: Context-Insensitive Code Example



```

Client's results...

'  %2 = alloca i32, align 4' could be: [] at   store i32 %unused, i32* %2, align 4
'  %2 = alloca i32, align 4' could be: [666] at   store i32 666, i32* %2, align 4
'data1' could be: [333] at   store i32 333, i32* %4
'data2' could be: [333] at   store i32 333, i32* %4
'  %1 = alloca i32, align 4' could be: [0] at   store i32 0, i32* %1
'data1' could be: [111] at   store i32 111, i32* %data1, align 4
'data2' could be: [222] at   store i32 222, i32* %data2, align 4
'data1' could be: [444] at   store i32 444, i32* %8
'data2' could be: [444] at   store i32 444, i32* %8
```

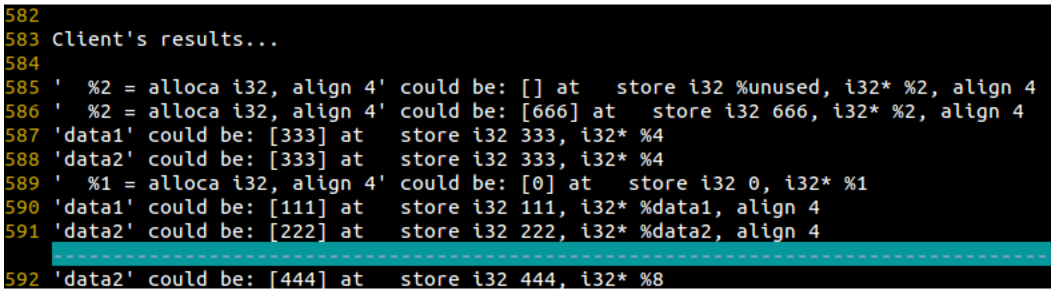
Figure 6: Context-Insensitive Smearing Arguments

## 5.2 Context-Sensitivity and Context Insensitivity

To show that our context-insensitive and context sensitive solutions functions properly, consider the code snippet in Figure 5.

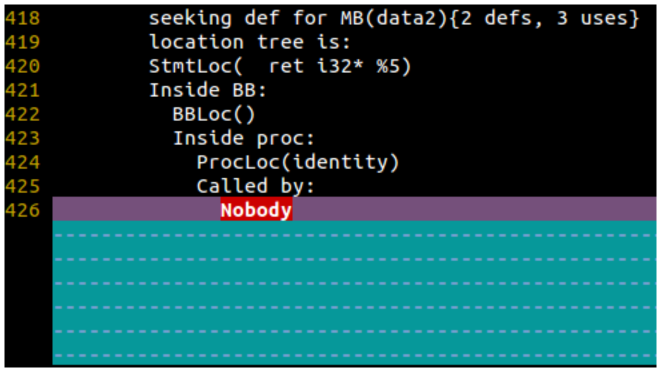
A context-insensitive solution will smear together arguments from calling functions, and also smear return values. It keeps track of only instance of the procedure and does not clone on every call. A context-sensitive solution, however, would clone the procedures on every call and keep a more complicated representation. The consequences of this are shown in Figures 6 and 7. The assignment at the end to the pointer `clone2` in Figure 5 combined with a context-insensitive solution would dictate that now either `ptr1` or `ptr2` can be 444. A context-sensitive solution, however, would not lose precision here.

In addition to that, a look into the bowels of our debugging output in Figures 8 and 9 will show that when looking for a def of `data2`, the location trees are significantly different for the context-sensitive and context-insensitive version. In particular, the context-insensitive



```

Client's results...

'  %2 = alloca i32, align 4' could be: [] at   store i32 %unused, i32* %2, align 4
'  %2 = alloca i32, align 4' could be: [666] at   store i32 666, i32* %2, align 4
'data1' could be: [333] at   store i32 333, i32* %4
'data2' could be: [333] at   store i32 333, i32* %4
'  %1 = alloca i32, align 4' could be: [0] at   store i32 0, i32* %1
'data1' could be: [111] at   store i32 111, i32* %data1, align 4
'data2' could be: [222] at   store i32 222, i32* %data2, align 4
'data2' could be: [444] at   store i32 444, i32* %8
```

Figure 7: Context-Sensitive Smearing Arguments



```
      seeking def for MB(data2){2 defs, 3 uses}
      location tree is:
      StmtLoc(  ret i32* %5)
      Inside BB:
        BBLoc()
        Inside proc:
          ProcLoc(identity)
          Called by:
            Nobody
```

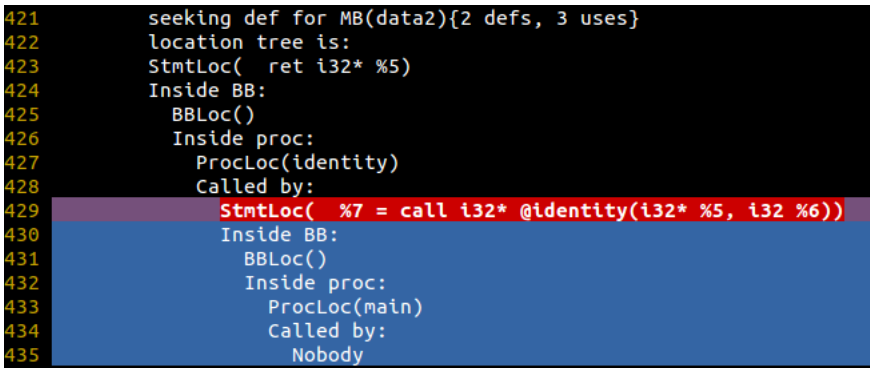
Figure 8: Context-Insensitive Location Tree

version does not have a "called by" portion like the context-sensitive version does.

## 6 Limitations

Due to time constraints, there are some limitations with the parts of the analysis that we did implement:

- Client analyses should be able to specify forwards or backwards analysis, but we are currently fixed at forwards.
- We have not tested our analysis on IR generated from anything other than "clang -O0". Theoretically any valid IR should work, but mapping C-level operations to LLVM IR is sometimes tricky. We had particular difficulties distinguishing pointer dereferences from pointer copies.
- On that note, we have only ever tested and reasoned about C. C++ and other source languages may work if the IR generated is similar.



```
        seeking def for MB(data2){2 defs, 3 uses}
        location tree is:
        StmtLoc(  ret i32* %5)
        Inside BB:
          BBLoc()
          Inside proc:
            ProcLoc(identity)
            Called by:
              StmtLoc(  %7 = call i32* @identity(i32* %5, i32 %6))
              Inside BB:
                BBLoc()
                Inside proc:
                  ProcLoc(main)
                  Called by:
                    Nobody
```

Figure 9: Context-Sensitive Location Tree

- Guyer's thesis mentions "multiple instance analysis", an IDFA performed to determine if a malloc must represent exactly one memory block at runtime. This permits strong updates on some heap blocks. We did not attempt this.
- Procedures must be analyzed conservatively in the presence of indirect function pointers and recursion. We detect these cases and do no analysis. This was not tested.
- Context- and flow-insensitive modes were not tested as heavily as the sensitive modes. In particular, we do not have any tests involving loops.

## 7 Future Work

We did not reach several major goals:

- integrate with client DFA by associating flow values with defs, as well as points-to sets
- creating defs and uses that annotations specify when we call a library function
- the Monitor and Adaptor. It was difficult to trace sources of imprecision before integrating with the client analysis.

Aside from these original goals, we have broader visions for extensions.

### 7.1 Refactoring

With many collaborators and a rapid development schedule, the code has outgrown our initial design sketches. There have been, to say it colloquially, hacks upon hacks that we have implemented to interface our code with one anothers. For instance there are globals littered in different files, and a need for a more sustainable debugging system. Problems as simple as differing naming conventions make portions of the code difficult to read.

More generally, the problem is that there are no unifying design principles guiding our code. We believe this is a priority before we move too much further, to ensure that our code does not become an unmaintainable mess.

In addition to simple refactoring, there are places where we are not fully using to our advantage the power of LLVM. We believed it would be simpler to work on our own code, but there already exists an `AliasAnalysis` interface in LLVM that we may have used [3] . Our knowledge of LLVM going into the project was very minimal, so using what we've learned we can harness more tools that LLVM offers. The `AliasAnalysis` interface may be wise to eventually implement if we open-source this work, so other clients can use our pointer analysis.

Now that we have a more sophisticated understanding of the architecture of the system, there are certainly changes we could make to improve clarity. One example where we chose to do this anyway was the list of phi merge functions per memory block. Guyer's thesis stores these per block, but the reference implementation moves these to the Procedure DB. This creates a chain of calls that is much harder to follow, so after we determined this structuring does not cause any different semantics, we chose the straightforward choice of storing phis in each block in our implementation.

## 7.2 Testing (at scale)

In the time that we had, we were unable to incorporate larger and more substantial tests. More than that, there is a need to ensure that the fundamental components work as expected. One improvement we plan to make in the future is to add in a more comprehensive testing suite, one that will allow us to verify correctness more simply. This will also allow us to then make changes and then verify that our changes worked with confidence. This is a high priority, and will be imperative as we move forward in this project.

We also had thoughts about automatically generating example C with randomized interleavings of procedure calls, branches, and levels of pointer dereference. In a higher level language operating at a much higher level of abstraction (since the code output is a function of the program data we generate), performing the same pointer manuever might be much easier to verify. This means we could compare the results between the two implementations automatically.

## 7.3 Open Source

It is our vision that one day others can benefit and improve upon the ideas that Broadway put forth. We plan to open source the project once it is in a good enough state, and let the community contribute changes. We developed a pass to split Basic Blocks at procedure calls so that placing virtual phi functions can be done more consistently. We also ported a pass to compute the ancestors of a function (all callers of it, flattened from the usual call graph). Both would be generally reusable.

## 7.4 Optimization

The LLVM Programmer's Manual recommends different implementations of sets and maps based on precisely what they store [1]. We chose to use STL data structures because they work immediately, while the LLVM specializations have poor documentation, no simple examples, and have a higher initial learning curve. A simple improvement is to use specialized structures instead.

## 7.5 Staged Analysis

There has been related work to perform F.S. pointer analysis much more quickly by performing a conservative, fast auxiliary analysis before the main pass [4]. Although Broadway does something similar, it still adds all reachable basic blocks to the worklist upon changes. It should be possible to merge these two ideas and create a much less redundant main pass that re-visits less blocks.

# 8 Appendix A: Build instructions

The entirety of the source code can be found at the following link: `https://bitbucket.org/aluong/broadway/`. Please let us know if you cannot get access to the repository.

## 8.1 Dependences

We assume the following about your system.

- You have compiled your own version of LLVM (living in `LLVM-ROOT-DIR`)
- You are working in a *nix environment
- `clang` is installed
- `mercurial` (`hg`) is installed

Wherever your system may deviate from this, adjust our instructions accordingly.

## 8.2 Obtaining the Source

To get our source code, ensure that you have mercurial installed on your system and run:

```
hg clone https://bitbucket.org/aluong/broadway
```

For simplicity, we would recommend you run this clone command from `LLVM-ROOT-DIR/lib/Analysis`.

## 8.3 Compiling

Once you have the code, open up `build.sh`. Edit the shell variable `$LLVM_RELEASE_TYPE` to match your release type. Then `chmod +x build.sh` if necessary, and run the script. It may spit out an absurd number of warnings, but that is okay. They are not from our code.

## 8.4 Running

We have a couple scripts present that we used when developing, that you can use to test/run the code. The script `testlogs.sh` will run all the c programs in the `tests/` directory and produce output for all of them into the `logs/` direcotry. The script `test_fi.sh` will run all the `passes*.c` tests in the `tests/` folder, and produce output in the logs; one file for a flow-insensitive run, and one file for a flow-insensitve run. And lastly, there is a `combinecommand` script that runs a taintedness test on `AnnotationPass/test/taintedness-test1.c`. There is also a `test_fi.sh`, but it has been failing some recent tests. If running opt manually, note that the `-split-at-call` pass must be run before the `-main` pass for our assumptions to hold. `addRequired<SplitPass>` did not seem to do what we wanted in that respect, but it does work this way.

## References

[1] Dinakar Dhurjati and Chris Lattner. *LLVM Programmer's Manual*, November 2011. http://llvm.org/docs/ProgrammersManual.html#datastructure.

[2] Samuel Guyer. *Incorporating Domain-Specific Information into the Compilation Process*. PhD thesis, University of Texas at Austin, 2003. http://www.cs.tufts.edu/~sguyer/publications/thesis.pdf.

[3] Chris Lattner. *LLVM Alias Analysis Infrastructure*, October 2011. http://llvm.org/docs/AliasAnalysis.html.

[4] Samuel Guyer Teck Bok Tok and Calvin Lin. Efficient Flow-Sensitive Interprocedural Data-Flow Analysis in the Presence of Pointers. 2006. www.cs.utexas.edu/~lin/papers/cc06.pdf.