Binding Time Analysis for Dynamic Compilation

Joel Auslander

Department of Computer Science and Engineering
University of Washington

Abstract

Dynamic compilation is a technique where some compiler optimizations occur dynamically (while the program is running) as opposed to statically (before the program starts). Dynamic compilation can speed up a program by optimizing it to take advantage of values that stay the same during execution, but that aren't known until after the program starts. An example would be options set on the command line. As long as the time saved by the optimized code is greater then the time spent optimizing, the result is a net win. The time spent optimizing can be greatly reduced if the static compiler prepares for dynamic compilation by analyzing the code to identify which operations can be optimized away and which can not. This analysis is called *binding time analysis*. Unfortunately, traditional binding time analysis only works for structured programming languages, such as lisp, and not for unstructured languages such as C. In order to efficiently dynamically compile C we have developed a new binding time analysis algorithm that works on unstructured programs. In addition, this algorithm fits into a standard framework for compiler algorithms called "iterative data flow analysis," which means it can be incorporated into any compiler that supports this framework. We incorporated binding time analysis into the Multiflow C compiler as part of our dynamic compilation system.

1 Introduction

When a program uses constants, a traditional compiler applies *constant-oriented optimizations* such as constant folding, full loop unrolling, strength reduction, and inlining of values into an instructions immediate field. These optimizations significantly speed up code that uses constants heavily. By comparison, code that uses variables is compiled under the conservative assumption that variables can take on any series of values during the program, which means that none of these constant-oriented optimizations can be applied. However, program variables typically don't all change at the same rate; often important variables will be constant for nearly the entire run of the program. It would be nice if variables that behave almost as constants could enjoy some of the performance advantages of constants.

We applied the tool of *dynamic compilation* to this problem. A traditional compiler runs before program execution--we call this *static compilation. Dynamic compilation* occurs while the program is executing, and can take advantage of information gained during execution to better optimize the program. With dynamic compilation, once the value of a nearly constant variable has been computed we can optimize code that uses it.

We focused on a specific type of nearly constant variable we call a run-time constant. A run-time constant is defined relative to a dynamic region, which is a region of code we intend to optimize at run-time. For a given dynamic region a variable is said to be a *run-time constant* if the variable has the same value every time the region is reached, *during any run of the program*. The first time a dynamic region is encountered we can evaluate all the associated run-time constants and use their values to compile an optimized version of the dynamic region. The dynamic compiler can treat the run-time constants as if they were true constants and apply constant-oriented optimizations. Because run-time constants never change after their dynamic region is reached, the optimized code will be correct for the remainder of the program's execution.

In order to produce high quality optimized code fast, we rely on aggressive precomputation during static compilation to set things up so that the final constant-oriented optimizations run as quickly as possible. We assume that the static compiler (with the programmer's help) can identify dynamic regions and run-time constants at static compile time. We use this information to separate each dynamic region into two parts: the part that computes run-time constants, and the part that uses them. The part that computes the run-time constants needs only be run once, since the values it computes never change. We call this part *setup code*. The rest of the code, called *template code,* contains references to the values computed in the setup code. In the template code we represent references to run time constants with special tokens called *holes*.

The setup and template code are then statically compiled all the way down to optimized machine code, except that the template code still has holes in it where it references run-time constants. Separating dynamic regions into template and setup allows run-time optimizations to go very quickly. To combine the template with the results of the setup the dynamic compiler just plugs run-time constants directly in the immediate fields of template instructions. To fully unroll loops the dynamic compiler stamps out copies of the loop body, plugging the holes in each copy with values specific to that iteration.

As an example, take the dynamic region

```
if (c>0)
  a = b+c;
else
  a = b-c;
x = y * a;
```

Now assume that `b`, and `c` are run-time constants--that they have the same value, for a given run of the program, every time this region of code is reached. That means that the static compiler can separate this region into the setup code

```
if (c>0)
  a = b+c;
else
  a = b-c;
rtc_1 = a;
```

and the template code

```
x = y * rtc_1;
```

Where `rtc_1` is called a *hole*, because it marks a location in the template code where the run-time constant `rtc_1` will be inserted during dynamic compilation. The setup code computes the value of `rtc_1`.

The static compiler further modifies the program so that the setup code is invoked only once, when the dynamic region is executed for the first time. The value of `rtc_1` is then compiled into the hole `<rtc_1>` in the template code. For example, if the setup code computed `rtc_1`=33 then the dynamic compiler would produce the optimized code

```
x = y * 33;
```

or even

```
x = y << 5 + y;
```

From then on, each time the dynamic region is encountered, the computer runs this optimized code, avoiding all the computations in the setup code.

Separating the original dynamic region into setup code and template code requires detailed information about which values are run-time constants, which loops are to be dynamically fully unrolled, which run-time constants are induction variables for these loops, and such. Ideally the static compiler would determine this information completely automatic, but unfortunately that's not currently feasible. Just choosing run-time constants that preserve correctness requires alias analysis and side-effect analysis that is very difficult to do in C. Choosing ones that actually lead to effective speedups is completely open research. On the other hand, it would be impractical for the programmer to specify all this information: there is a lot to specify, mistakes lead to subtly incorrect behavior, and most of the values involved are temporaries that can't even be meaningfully named by the programmer.

We solved this problem with a compromise in which the programmer gives a small set of annotations that supply the compiler just enough information that it can deduce the rest on its own. Specifically the programmer tells the compiler the boundaries of the various dynamic regions, specifies which variables hold run-time constants on entry into those regions, and specifies which loops should be completely unrolled. There is also a special annotation needed to deal with complex structures that are only partially run-time constant. The compiler then deduces additional implied run-time constants by noting, for example, that result of adding two run-time constants is a run-time constant.

My main contribution to the project was the design and implementation of the *binding time analysis* that takes programer annotations and computes for every variable in the region whether it is a run-time constant or not, and simultaneously determines which basic blocks are contained in which dynamically unrolled loops. I implemented this analysis in a compiler pass that then uses the information to separate the dynamic region into setup and template code. This pass runs as an early stage in the pipeline for Multiflow, a top-of-the-line research compiler. The setup and template code then continue through the pipeline, undergoing the traditional optimizations. At run time the setup code executes normally, computing the values run time constants. These values, along with the compiled template code, are passed to the dynamic compiler, which then generates the final optimized code for the dynamic region.

2 Binding Time Analysis

2.1 Roadmap

The preceeding section gave a basic overview of what binding time analysis is (determining run-time constants) and why it's useful (it allows you to prepare for dynamic compilation statically so that the run-time cost is low). This section goes into the details of what precisely the input and output of binding time analysis is, and how it's computed.

I'll start with the simple case: binding time analysis on structured code. Structured code is code that can be represented by a tree of if's, switch statements, and loops. In languages without goto's, such as lisp, all code is structured. Binding time analysis is particularly easy for such code, which is one reason some other dynamic compilation systems are based on lisp or a similar language.

Because we are targetting C and other languages with goto's, our binding time analysis must handle unstructured code as well. When analyzing unstructured code you start with less information about how control flows throught the code then when analyzing structured code. Some of this information is necessary for binding time analysis. So where in the structured case we got this information for free, in the unstructured case we have to explicitly compute it, as a precursor to identifying run-time constants. This makes the unstructured analysis significantly more complicated than the structured one.

The critical piece of information that is missing in unstructured code is the relationship between places where control flow splits (if’s or switch statments) and the places where control flow merges again. In structured code there is a direct connection between every split and and associated merge, and its trivial to compute which splits correspond to which merges.

We need this information to distinguish between two kinds of merges: static merges and dynamic merges, a distinction I’ll define later. In structured code we can identify whether a merge is static based on properties of its associated split. This is why the pairing of merges and splits makes structured code easy to analyze.

In unstructured code the merges and splits don’t pair up in any nice way. The relationships between splits and merges are inherently more complicated. Here it becomes difficult to even define a static vs. a dynamic merge, let alone identify one. Unfortunately we have to, in order to do binding time analysis.

We solve this problem with a new concept, called reachability conditions. Reachability conditions is a new program property that can be defined even for unstructured code, and that captures enough of a relationship between splits and merges to allow us to identify static merges. We compute reachability conditions with a new analysis that forms part of the complete binding time analysis, enabling us to identify what really matters: the run-time constants.

Static merges, reachability conditions, and the analysis that computes them are complicated enough that I’ll explicitly gloss over them while I explain the basic analysis for unstructured programs. Instead I’ll use a simpler definition of static merges that covers the example code but needs to be extended to handle general code. I’ll explain static merges and reachability conditions fully in their own subsections, and show how they can be computed as part of the larger analysis.

Neither of the first two examples I’ll use contain any loops. In general, most of the algorithm is captured without loops; they just add some minor complications which I’ll address after the basic algorithm has been specified. First, loops make the analysis trickier because they can introduce circularities that make it difficult to analyze the program a piece at time. This is a common problem in program analysis in general, not one particular to our binding time analysis, and also more of an implementation problem than a conceptual problem. I'll address this when I talk about the implementation of our analysis.

The second problem is more central to binding-time analysis: dynamically fully unrolled loops. Remember that binding time analysis is intended to help the compiler prepare for dynamic compilation. One of the optimizations that can be applied at dynamic compile time is to fully unroll a loop whose bounded by a run-time constant.

In the final part of this section I'll specify our analysis formally as an iterative dataflow analysis or IDFA. IDFA is a general framework that covers a wide range of analysis. By showing how to express our analysis in this framework we can both give a very precise specification of exactly how it works, and we can take advantage of general results apply to all IDFA's. For example, our implementation is based on a general algorithm that applies to any IDFA Because this algorithm has been studied in general we can use these results to deduce both that the algorithm computes what we intend it to and to reason about how long it takes to run.

To summarize, the roadmap for this section is as follows

- define the problem binding time analysis solves
- show how to do BTA for structured code by analyzing a simple example
- give the general approach to analyzing unstructured code

At this point we get into the nitty-gritty of what precisely happens during the analysis of each basic block. The next few subsections:

- give the basic binding time analysis for control flow graphs, temporarily ignoring how we identify (or even define) static merges
- define static and dynamic merges in general
- define reachability conditions and show how they can identify static merges
- show how reachability conditions are computed
- describe how to handle loops

This completes the description of the algorithm. The final subsection I fit this analysis into the Iterative Data Flow Analysis framework. This allows me to give an algorithm for implementing the binding time analysis, and to reason about its running time..

2.2 The Problem

Binding time analysis takes annotated source code and determines, for every value in a dynamic region, whether it is a run-time constant or not. Remember that a run-time constant is any variable in the dynamic region that has the same value every time it is used, during a single execution of the program. It would be too much to ask for the binding time analysis to be 100% accurate, in fact the problem is incomputable. Instead we demand that our binding time analysis be conservative, so that it never claims a variable is a run-time constant when it isn't. Then we've tried to design our analysis to be as accurate as possible.

The input to the binding time analysis is the program code with annotations. The annotations tell the system where dynamic regions begin and end, and for each dynamic region what variables are known at the entry point to be run-time constants. The annotations also specify which loops are to be fully unrolled dynamically (these must be loops whose bounds are run-time constants). The binding time analysis then deduces which values inside the dynamic region must also be run-time constants.

In keeping with the terminology of other dynamic compilation systems, we call variables that are known to contain run-time constants "static" and the rest "dynamic." The output of the binding time analysis is then a labelling of the program that specifies at each point in the program which variables are static and which are dynamic. More precisely, we compute a static set at each point that contains which variables are known to be static, and any variable not in that set is assumed by default to be dynamic.

As an example, given the annotated source

```
dynamic_region(x) {
  t = x + 1;
  u = t + y;
  if (x>0)
```

```
    v = x;
  else
    v = -x;
}
```

The binding time analysis should deduce at the end of the region that `t`, x, and `v` are static but that `u` is dynamic. This is because for any fixed value of x (e.g. x=-17) the values of t and v are also fixed (t=-16, v=17) but because y can vary, the value of u isn't fixed. The programmer can tell this just by looking at the code, but the compiler has to use a systematic analysis.

2.3 Analyzing Structured Code

Because this code doesn't contain any goto's, it can be represented in a structured form. We can then analyze it with just a few simple rules:

- At the start of the region we assumed that the only static variables are those specified by the annotation, in this case only `x`.
- If a variable is defined as the result of an operation (such as + or -), it's static if and only if all the operands are static (or constants). This is why `t` is deduced to be static but `u` isn't.
- If a variable is defined inside an `if` statement you use the rule that its only static after the `if` when both the branch predicate is static and when along either branch the variable is defined statically. In the case of a nested `if`'s this rule gets applied recursively. In the above example, if the branch predicate were `(y>0)` instead of `(x>0)`, or if on the true branch the statement were `v = u` instead of `v = x`, then `v` would be dynamic after the `if`.
- A similar sort of rule holds for values defined inside loops. Basically if the loop bounds are static, and a variable is defined statically on each iteration of the loop, then after the loop the variable can be deduced to be static.

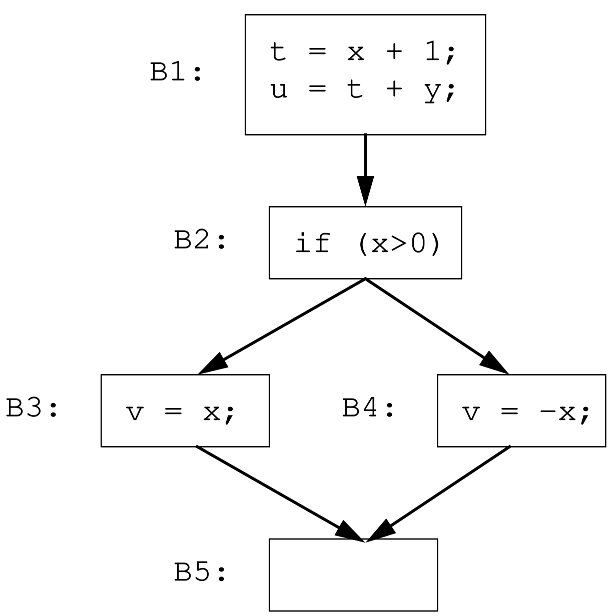
This approach to binding time analysis assumes that your program is represented in terms of nested if's and loop's, and so it only works for structured languages code.

2.4 Analyzing Unstructured Code

There are two major changes necessary to extend binding time analysis to potentially unstructured graphs. First off, unstrucutred code is represented differently than structured code, inside a compiler.

Structured code is typically represented as a tree, called an abstract syntax tree, where the nesting of tree represents the nesting of the program. Because in unstructured code control can potentially flow from any point to any point, unstructured code is represented as an arbitray graph called a Control Flow Graph or CFG.

A CFG represents a program as a graph whose nodes are basic blocks and where two nodes are connected with a directed edge if program control can pass from the first node to the second. A node only has more than one outgoing edge if its a branch or switch statement. The CFG corresponding to the simple example is given by:



The second major difference between analyzing structured and unstructured code is that the binding time analysis now has to compute more than just static sets.

In the end the static sets are what we really care about, but it turns out that in order to compute them we have to compute other information as well. To do binding time analysis we have to know things like where a static variable was last defined and when a basic block (here B5) merges control that split at a static branch (here B2). Before these properties were obvious from the structured representation of the code. When analyzing a CFG, we have to compute such information as part of the analysis, before we can compute the static sets.

We therefore add two additional bits of information to the binding time information we compute at each program point: definition points and reachability conditions. The definition points specify for each static variable where it was last defined. In the example the definition points at B3 would specify that t had been defined at B1.

Reachability conditions are more complicated. They are used to identify static merges. For now we can define a basic block to be a static merge if it ends an if statement whose predicate is a run-time constant. In this example B5 is a static merge because it ends the if statement that begins at B2, and the predicate (x>0) is a run-time constant. Actually this definition is more narrow then it has to be: there are merges that don't fit into this definition that can also be considered static merges. We'll give a broader definition of a static merge later, and use it to motivate reachability conditions, which can identify general static merges.

However its not necessary to know all these details right now. For now we'll just use the simple definition of a static merge and ignore how the analysis identifies them. We'll just focus on the other parts of the binding time analysis: static sets and definition points. Ignoring reachability conditions, the binding time analysis just computes at each program point a set of static variables labelled with definition points. For example, if the binding time analysis computed the information {x:B0,t:B1}, it would mean that at that program point that x and t are static, and x was last defined at B0 while t was last defined at B1.

2.5 Binding Time Analysis for a Control Flow Graph

Remember that binding time analysis works on a CFG by starting at the entry point and then visiting each basic block in turn, computing binding time information as it goes. The binding time information at the entry point is computed based on the annotations at the start of the region. For the other basic blocks, the BTA computes binding time information at each block based on the binding time information of the basic blocks that come before it.

The rules for computing static sets while analyzing a CFG are basically the same as those for strucutred code, just translated into the CFG framework. This makes sense, since in both cases the analysis is computing the same thing.

First we need the rule for computing the static set on entry to the region:

- The only static variables at the entry point (`B1`) are those specified in the annotation (only `x`).

We'll consider them to be defined at some `B0` from before the region.

Next we need the rule used for computing the static set at the end of basic block based on the static set at the beginning of the basic block. This rule is applied to each statement in the basic block in turn:

- After a statement, the variables defined by the statement are static if and only if all the operands were static or constant. The static variable is labeled as having been defined at that statement (in these examples I've been using the basic block labels, to keep things simpler, but in reality you use a different label for each statement)..

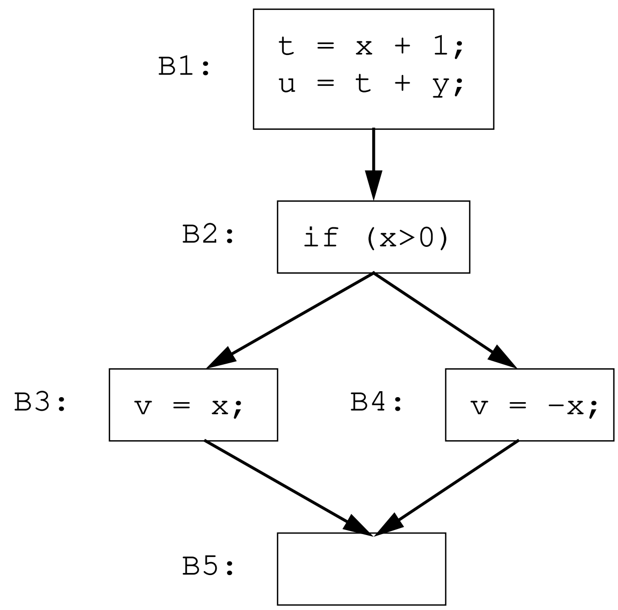
Whatever static set is computed to be valid at the end of a basic block is propagated along each edge out of the basic block. Similarly, the static set at the start of a basic block is computed based on the static sets of all the incoming edges. If there is only one incoming edges, things are simple: that start of the basic block is labeled with whatever static set is on the incoming edge. When the basic block has more than one incoming edge, then it is a merge. When a basic block is a merge, the BTA combines the static sets of all their incoming edges to compute the static set at the start of the basic block. The rule the BTA uses to combine the static sets depends on whether the merge is a static merge or a dynamic merge.

We basically translate the third rule (the one about if's) from the simple analysis into a merge rule for the CFG analysis. After a static merge, a variable is static if it is in the static sets propagated on all the incoming edges. If the variable is tagged with the same definition point in all those static sets, then it should be tagged with this definition point after the merge as well. If its tagged with different definition points, its considered to be defined by the merge.

A variable is static after a dynamic merge only if it is not only static at all predecessors but its also tagged with the same definition point at predecessors. This basically means that whatever happened between the point at which the variable was last defined and the point at which the merge occured didn't affect that variable, so if the variable was static before, its still static now.

As a simple example of how this works in practice, we can re-analyze the structured example using the CFG representation. We should get the same answer we did before, since the run-time constants don't depend on how we represent the code or how we do the analysis, but are a property of the code itself. This example shows that the unstructured analysis truly extends the structured one. It also is instructive in general, since the same approach applies to any code that can be represented as a CFG, since we never rely on any properties of structured code while analyzing the example. Later, when I define general static merges and reachability conditions, I'll analyze a truly unstructured example with the same techniques used here.

Remember the CFG:



Analysis starts at the entry point B1. The binding time information here is given by the annotations at the start of the region. Remember that they specify that on entry the only variable known to hold a run-time constant is x. This leads to the binding time information {x:B0}, where B0 is just some basic block associated with the start of the region, where the annotations occur.

The BTA then takes this initial condition and looks at the statements in block B1, from which it deduces that after B1 the binding time information is {x:B0,t:B1}, since t is defined as the sum of the run-time constant x and the constant 1.

The BTA then analyzes B2, and since there is only one incoming edge the information at the start of B2 is the same as at the end of B1: {x:B0,t:B1}. This information is unchanged in B2, so it will be propaged out to the start of both B3 and B4.

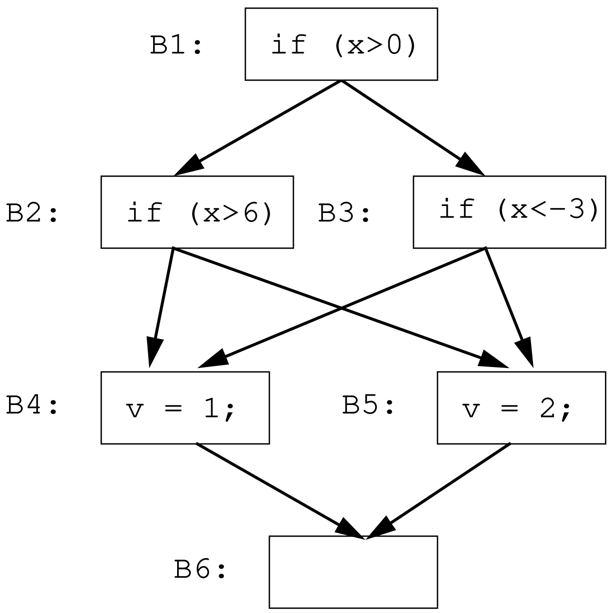
Next the BTA analyzes B3, and determines that since the incoming information is {x:B0,t:B1}, the outgoing information is {x:B0,t:B1,v:B3}. Similarly for B4 it takes the incoming information {x:B0,B1} and deduces outgoing information {x:B0,t:B1,v:B4}.

Finally the IDFA analyzes B5, which has two incoming edges; hence B5 is a merge. What the BTA does here depends on whether the merge is static or dynamic, just as in the structured version of the analysis the static set after the if depended on whether the if predicate was static or dynamic. In this case the merge is static, so the merged information is {x:B0,t:B1,v:B5}. v labeled with B5 because it's considered to be defined by the merge, since it was defined differently on the two incoming edges. Had the branch at B2 been predicated on a dynamic variable, then B5 would have been a static merge, and BTA would compute at B5 the binding time information {x:B0,t:B1}.

If you compare this to analysis of the same graph when it was represented as structured code, you'll see that the IDFA based BTA determines the same static sets, as it should. This example also shows why the BTA has to label variables with where they were defined: after a dynamic if, static variables that were defined inside the if become dynamic, but static variables that were defined before the if remain static (the if doesn't affect them). These definition points are needed to distinguish the two.

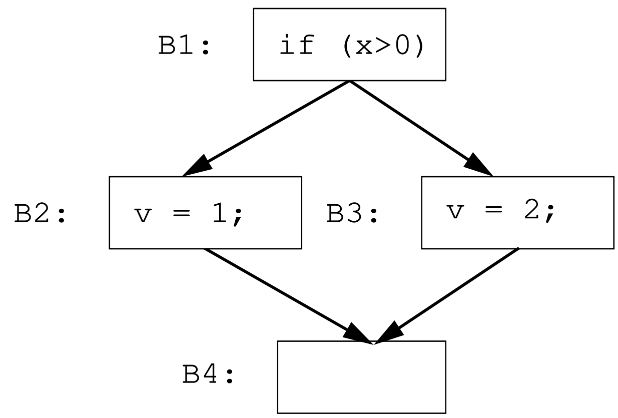
2.6 The General Definition of Static Merges

Now we can start to fill in the gaping hole in the analysis as described so far: how do you determine whether a merge is static or dynamic. The definition given above only works when you can associate the merge with a branch in such a way that the merge ends the if that the branch starts. There are two problems with this approach: first, even if such an association exists, how does the compiler figure out which branch goes with which merge? The deeper problem is that for general (unstructured) CFG's such an association between branches and merges doesn't always exist! Take as an example the unstructured CFG:



The merge at B6 doesn't really correspond to any of the branches in this code. So is it a static or a dynamic branch? In other words, which merge rule should we use to compute the static set at B6?

In order to come up with a definition of a static vs. dynamic merge that extends to all CFG's, we should look at what makes one kind of merge different from another. Let's look at the structured case again:



If x is constant then the branch at B1 will always go the same way. This means that only one of the edges going into B4 will be reachable. Because of this, it doesn't matter that v is defined inconsistently on the two branches, because only one of those branches will actually be taken. This leads to the merge rule for static merges. If x isn't constant, then on different executions of the dynamic region control will sometimes go one way, sometimes the other. This means v will have different values after the branch for different executions of the dynamic region. This leads to the merge rule for dynamic merges.

We use this as the basis for our general definitions of static and dynamic merges: a merge is static if, for any given execution of the program, control always flows down one and only one incoming edge. Otherwise the merge is dynamic. This is similar to the definition of a static branch: when you have a branch that depends on a static variable, every time that branch is reached, control always goes the same way. When you have a dynamic branch, control may go different ways. In a static merge control always comes from the same incoming edge. In a dynamic merge it could come from either edge. Under this definition the merge at B6 in the unstructured example is static, since for a fixed value of x, control will always flow down the same edge when entering B6.

With B6 known to be a static merge, we can now analyze the unstructured example.Again we are assuming that on entry x is static (this would be specified by the annotations at the start of the region). The first two rules determine that at the end of B4 the static set is {x:B0,v:B4} and at the end of B5 the static set is {x:B0,v:B5}. Now we can use the static merge rule to deduce that at B6 the static set is {x:B0,v:B6}. We use B6 to represent the last definition point of v because the value of v is effectively being computed by the merge--which value it has depends on from which direction control flowed.

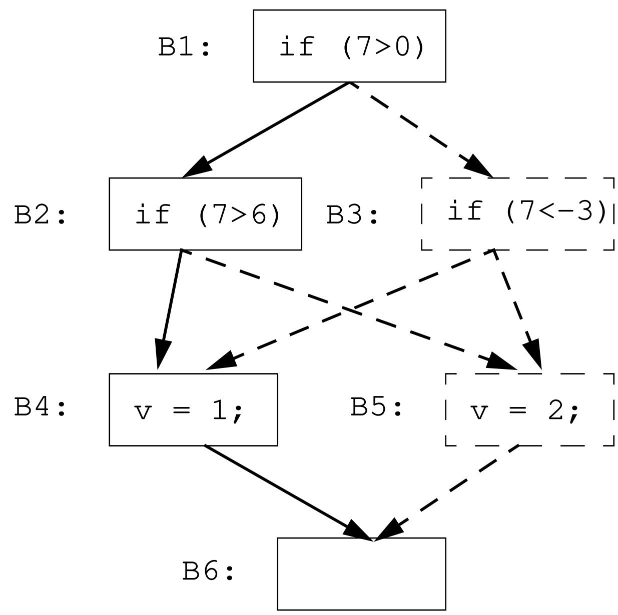
{x:B0,v:B6} is right answer, since, for any fixed value of x this code computes a fixed value of v. Specifically, v is 1 if (x>6) or (0<=x<-3), and v is 2 if (x<-3) or (6>=x>0). So in this case at least the analysis is accurate. There is still gap in the algorithm however: how precisely does the compiler compute, at a merge, whether it is static or not. In other words, how does the compiler deduce that for any execution of the program control will always enter a merge the same way?

2.7 Reachability Conditions

Our approach is based on the concept of reachability conditions. Reachability conditions capture under what assignments of values to run-time constants an edge in the CFG is reachable and under which assignments its not. A merge can be deduced to be static if the reachability conditions on the incoming edges are such that under *any assignment of values* to run-time constants, precisely one of the edges is reachable. Another way to say this is that a merge is static if there exists no assignment of values to run-time constants such that two of the incoming edges are both reachable simultaneously. If we can't use reachability conditions to prove that a merge is static, we assume that it is dynamic. This is a conservative approach in that it will never identify a dynamic merge as static but it may sometimes identify a dynamic merge as static.

The idea here is that assigning values to run-time constants simplifies the CFG. During any given run of the program, the values of all the run-time constants are fixed. Once this happens, static branches become effectively goto's. One of the outgoing edges of each branch becomes unreachable, bedclothes branch always goes the other way. So assigning values to run-time constant effectively removes edges from the CFG. After these edges are removed, additional code and edges become unreachable, and are effectively removed as well. A merge is static, for this assignment of values to run-time constants, if in this simplified graph there is only one incoming edge.

For example, imagine that during some execution of the program containing the unstructured example we have "x=7." The CFG simplifies to



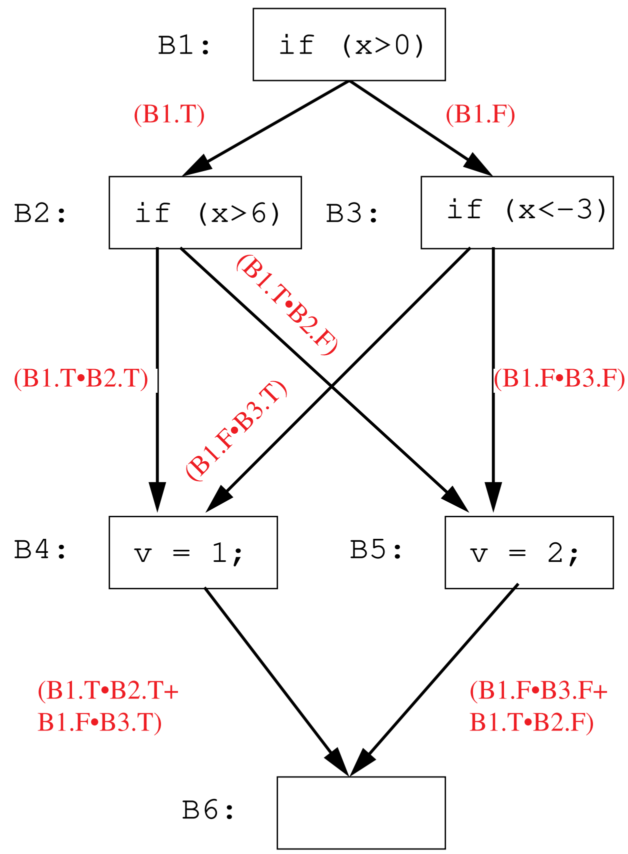
So that there is only one edge coming into the merge at B6. This is what we would expect, since the merge at B6 is a static merge.

Of course we won't know until run-time what assignment will be made and which edges will be removed, so instead we base our analysis on finding merges that we can prove are static *no matter what assignment of values to run-time constants actually occurs at run-time*. We do this by computing reachability conditions, which tell us under which assignments which edges become unreachable.

Any assignment of values to run-time constants removes some of the edges leading out of static branches. A reachability condition is a boolean expression that tells you which edges must remain in order for a given program point to be reachable. If we associate each edge leaving a static branch with a boolean variable (e.g. B3.F for the false branch leaving B3) that is true when that edge is present, then we can write reachability conditions as formulas over these variables.

We further simplify things by writing them in disjunctive normal form. So if a point in the program is reachable when the CFG contains the true branch out of B2, or when it contains both the false branch out of B4 and the true branch out of B6, we could write this as (B2.T+B4.F•B6.T)

So if we label the unstructured example with reachability conditions, we get:



Once we have the reachability conditions, we can use them to identify static merges. In this example, the reachability conditions on the edges incoming at B6 are (B1.T•B2.T+B1.F•B3.T) and (B1.F•B3.F+B1.T•B2.F). The question is, are these two sets of conditions ever simultaneously satisfied? This is equivalent to asking whether there is a solution to their logical conjunction, (B1.T•B2.T•B1.F•B3.F+B1.T•B2.T•B1.T•B2.F+ B1.F•B3.T•B1.F•B3.F+B1.F•B3.T•B1.T•B2.F).

This formula looks a lot more complicated then it really is. We can simplify formulas two ways. First, we can remove any clause with both a true edge and a false edge from the same static branch, since both will never be present at the same time. This means eliminating any clause containing (Bx.T•Bx.F) for any branch x. Similarly, we can factor out terms of the form (Bx.T+Bx.F) since this expression is always true. In this case only the first rule is needed eliminate all the clauses in the formula and simplify it to (FALSE).

2.8 Computing Reachability Conditions

Once we have the reachability conditions, we can distinguish static and dynamic merges, which means we can do binding time analysis. But how do we compute reachability conditions? We use an approach similar to that we used for binding time analysis: we give rules that specify the binding time conditions at the top of the graph, and how those binding time conditions change across statements and at merges.

- On entry the reachability conditions are (TRUE), since the entry is always reachable.
- If the reachability conditions at the start of the static branch Bx are given by the formula f, then on the outgoing true edge the reachability conditions are f•Bx.T and on the outgoing false edge the reachability conditions are f•Bx.F.
- On the outgoing edges of any block that isn't a static branch the reachability conditions are the same as they were going in. This is true even for a dynamic branch. The outgoing edges are reachable precisely when the basic block.
- At a merge, where the incoming edges have reachability conditions given by formulas f1,f2,...,fk, the reachability conditions after the merge are given by f1+f2+f3•••fk.

For the most part these rules are all it takes. If you look at the example, you'll see that at each program point these rules tell you how to compute the reachability conditions on outgoing edges from those on incoming edges.

There is one more important technical detail however. The rules for computing reachability conditions assume you know which branches are static and which are dynamic, which assumes you've done binding time analysis. However you can't do binding time analysis until you have reachability conditions! We break this circularity by computing binding time analysis and reachability conditions simultaneously.

As we traverse the CFG and encounter each basic block in turn, we first use the reachability conditions on the incoming edges to determine whether there is a static merge or not. Then we can compute the static set on entry to the block, and determine whether that basic block is a static branch or not. Then we compute the reachability conditions on the outgoing edges. This process of using reachability conditions to compute static sets and using static sets to compute reachability conditions continues until we've computed both for every program point in the dynamic region.

2.9 Loops

The analysis described up to know works for regions without loops. There are a couple technical details involving loops that must be covered before you have a complete analysis.

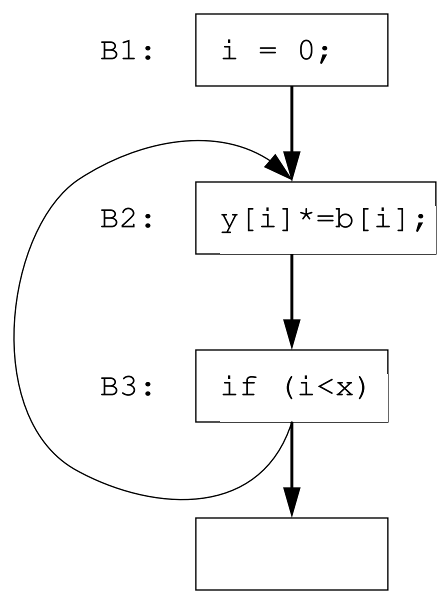
First there is the problem of how to do the analysis. When a CFG has no loops you can traverse the basic blocks in such an order that you have always computed the binding time information (static sets and reachability conditions) of the block's incoming edges before you analyze the block to determine the binding time information on its outgoing edges. There is no problem here.

When the region has a loop this isn't possible. No matter how you traverse the graph you will always run into a situation where you have to analyze a basic block before you know the binding time analysis on all its incoming edges. The solution is to *guess* what the binding time analysis on some of the incoming edges to a basic block is. Later on, when you've traversed the graph, you can check whether what you finally compute for that edge is the same as what you guessed. If its not, you have to revise the guess and start the analysis over from that point. This kind of approach is called iterative data flow analysis, and we'll talk about it in more detail in the next section.

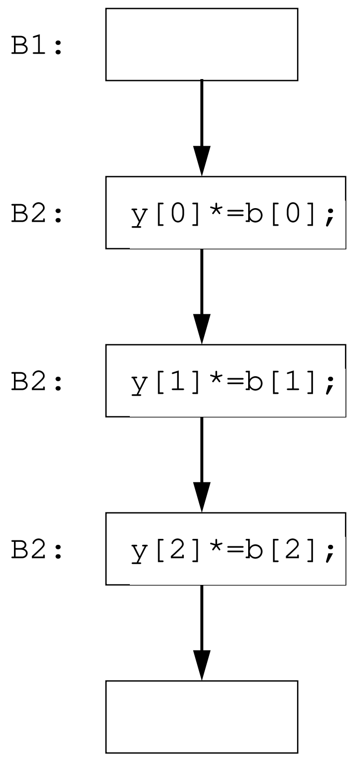
The next issue is more substantial: dynamically unrolled loops. We support the run-time optimization where a programmer can specify a loop to be fully unrolled during dynamic compilation. The programmer specifies this with an annotation. Full loop unrolling means that during dynamic compilation the loop is replaced with N copies of the loop body (where N is however many times the loop iterates). Within each loop body any induction variables of the loop are run-time constants. This is only a valid annotation when the program is certain that N, the number of times the loop is iterated, is a run-time constant.

In order to support dynamic loop unrolling we need to extend our analysis in several ways. First, we need to find the loops in an unstructured CFG. For this we rely on a separate analysis, built-in to the multiflow framework, which identifies loops and determines a loop head, loop back branches, loop entries and loop exits. Next we need to modify our static merge rule. At the head of a loop the loop back branches merge with the loop entry. The reachability conditions indicate that this a dynamic merge, since the back branches are only reachable when the entry branch is. For normal loops this lead to the correct deduction that loop induction variables are always dynamic.

For a dynamically unrolled loop the binding time analysis always treats the merge at the head as static, regardless of what the reachability conditions say. This makes sense because during dynamic compilation the loop



becomes, for x=3:



Now there as many copies of the loop head, B2, as there are iterations. However each copy has only one incoming edge. So its really a static merge.

When we treat the heads of such loops as static merges, the binding time analysis correctly determines that the loop induction variables are run-time constants.

2.10BTA as an Iterative Data Flow Analysis

The analysis as described so far can be rephrased as an Iterative Data Flow Analysis in a straightforward way. The lattice is just the cross product of the three parts of the binding time information: the static sets, the definition points, and the reachability conditions. Each of these can be mathematically formalized in a simple way: static sets are just sets of variables, definitions points are just functions whose domain is a static set and whose range is the set of program points, and the reachability conditions are just sets of sets of CFG edges which represent boolean formulas written in disjunctive normal form.

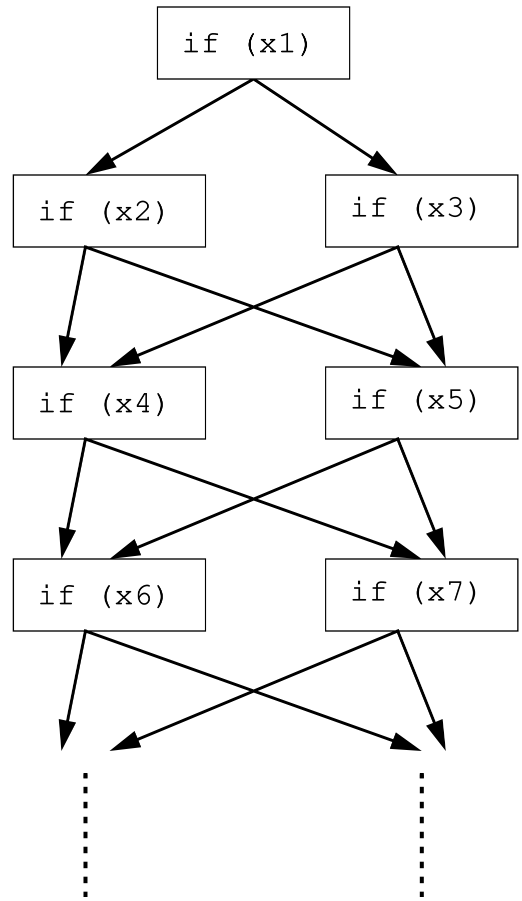
Once the lattice is formalized, the rest can be formalized as well. The initial conditions can be rewritten in mathematic notation. So can the flow functions, which are just the functions that specify how the binding time changes across a statement. The merge rule can similarly be rephrased in mathematical notation.

Once this is done we find that the lattice and the functions indeed satisfy the requirements of an IDFA.

The lattice is a partial order, based on the following: lattice element x is less than or equal element y if x's static set is contained in y's static set, and if x's reachability conditions are implied by y's reachability conditions. Top has everything static but is never reachable, and bottom as nothing static and is always potentially reachable.

The merge function now always result in a resulting lattice element that is less than or equal to all of the incoming lattice elements. This is because the static sets are interesected and the reachability conditions are unioned.

This means that the standard IDFA algorithm, which is the one I implemented, will eventually converge. Unfortunately, in theory at least, this convergence can take exponentially long. This is because the reachability conditions theoretically can grow exponentially long in the number of edges in the CFG. Take for example the following CFG, for which the reachability conditions double in size with each level:



While this means that in theory the binding time analysis can take vast amount of time, in practice it has always run quickly on the test cases we have given it. In fact the construct above is quite unusual. Normal programs are mostly structured, and in a structured program the reachability conditions don't grow very big. This is because after a merge in a structured program the merged reachability conditions always simplify to what they were before the corresponding branch occured. Also in a structured program reachability conditions will only have one clause (they are a given by a single logical disjunction). Since this single clause only grows by one term at each branch, and always shrinks by at least term after each merge, it is never larger then the nesting depth of the program--which typically isn't very deep.

If, however, we were to someday run into a class of extremely unstructured programs that actually caused the binding time analysis to blow up, there is a simple fix we could implement. At no point do we assume that our reachability conditions are precise, instead we only assume they are conservative. That is we assume that a program point is only reachable when its reachability conditions are satisfied, but we don't assume that it definitely is reachable whenever its reachable conditions are satisfied. Because of this, we can always replace reachability conditions that have grown too big with a simpler set of conditions that subsumes the original.

This won't make the analysis incorrect, it will just make it less precise, because now some static merges will appear to be dynamic merges, and potentially some run-time constants won't be detected. Therefore, if exponentially large reachability conditions ever become a problem, we can always set a limit on how large they can grow. This will limit the running time of the algorithm, at the cost of accuracy.

3 Transforming the Program

Binding time information computes static sets for every point in the program. This isn't an end in itself, however. This binding time information is only computed so that it can be used by the compiler to transform the original program. The compiler uses binding time information to separate the original code into setup and template code. This transformation allows us to later do dynamic compilation quickly.

The basic idea is that setup code is the code that computes run-time constants, and template code is everything else. In the original region, these two types of code are intertwined. Run-time constants are computed by the setup code and then used by the template code. In order to separate them we have to introduce the concept of holes, markers for places where template code uses run-time constants.

Once we've split the code apart, it can be run in phases: first the setup code runs to compute run-time constants, then the template code runs and, at every point where the template code has a hole, it uses a run-time constant computed by the setup. Because run-time constants never change, we actually need only run the setup code once. What ever values it produces are valid for the rest of that run of the program. The dynamic compiler can therefore optimize the template code by replacing the holes with whatever values were computed by the setup code.

For example, take the code, which evaluates a polynomial of degree n at the point 1+x, where n and x are run-time constants (but the polynomial possibly changes over time):

```
dynamic_region(x,n) {
    y = 1 + x;
    e = 1;
    s = 0;
    unrolled for (i=0;i<n;i++) {
        s = s + a[i]*e;
        e = e * y;
    }
}
```

Binding time analysis has already run and has taken the annotations, shown in green, and found the run-time constant shown in red. This code would be split into the setup code

```
y = 1 + x;
e = 1;
rtc_1 = 0;
for (i=0;i<n;i++) {
    rtc_2:l = i;
    rtc_3:l = e;
    e = e * y;
}
```

and the template code

```
s = rtc_1;
loop:l {
    s = s + a[rtc_2:l]*rtc_3:l;
}
```

Here blue denotes holes. The values of the holes are computed by the setup code and used in the template code.

This example shows some details that are particular to programs with dynamically unrolled loops. The holes `rtc_2:1` and `rtc_3:1` are defined inside such a loop (in terms of the BTA, this can be detected by their definition points). They'll take on a different value for each iteration of the loop. In the template code the for statment has been replaced with a special loop marker. The original loop calculations are no longer needed in the template code. Instead, the static code executes the loop to compute how many times it iterates. Then, during dynamic compilation, the template version is unrolled that many times.

So if a program containing this code were executed, and the dynamic region was reached with x=2 and n=3, then the setup code would compute the run-time constant values

```
rtc_1 = 0;
rtc_2:0 = 0;
rtc_3:0 = 1;
rtc_2:1 = 1;
rtc_3:1 = 4;
rtc_2:2 = 2;
rtc_3:2 = 16;
```

The dynamic compiler would then optimize the template code based on these values, to produce the final executable code:

```
s = 0;
    s = s + a[0]*1;
    s = s + a[1]*4;
    s = s + a[2]*16;
```

or, depending on how much time the dynamic compiler decided to invest in optimizing, perhaps even:

```
s = a[0] + a[1]<<2;
s = s + a[2]<<4;
```

Once binding time analysis has run, producing template code is as easy as it looks: the compiler just eliminates all the operations that compute run-time constants, leaving holes and loop markers as necessary. We call these removed operations static operations. The result is template code.

3.1 Computing Setup Code

Setup code, unfortunately, is a considerably harder to generate. Since all setup code does is compute run-time constants, and the operations that produce run-time constants are already present in the original code, it might seem trivial to generate setup code. Its true that setup code is made up of precisely the static operations that are removed from the original program when producing template code. That part is easy. What's hard is how to connect these operations together with the proper flow control.

To see why this is non-trivial, take the following example:

```
dynamic_region(x) {
    if (x>0)
        u = 1 + x;
    else
        u = 1 - x;
    if (b)
        v = 2 * x;
    else
        v = 5 * x;
}
```

This example produces the template code:

```
    u = rtc_1;
    if (b)
        v = rtc_2;
    else
        v = rtc_3;
```

and the corresponding setup code:

```
if (x>0)
    rtc_1 = 1 + x;
else
    rtc_1 = 1 - x;
rtc_2 = 2 * x;
rtc_3 = 5 * x;
```

The first if statement dependes on a run-time constants, and therefore for any run of the program only one of the two branches will be taken. In this case the entire branch is moved into the setup code. When the setup code is run the if executes and computes `rtc_1` one of the two possible ways.

But in the second case the if depends on a dynamic value, since b hasn't been annotated as a run-time constant. In this case the if stays in the template code, and during a single run of the program the branch may sometimes go one way and sometimes the other way. In this case there are two run-time constants, one for each branch, and the setup code must always compute both of them.

If all code were structured then these two approaches could be applied in general. We could analyze each if statement, see if it was based on a static or a dynamic predicate, and use one or the other approach accordingly. Unfortunately, just as with BTA, we can't depend on code being structured. We have to be able to produce setup code for arbitrarily crazy control flow graphs.

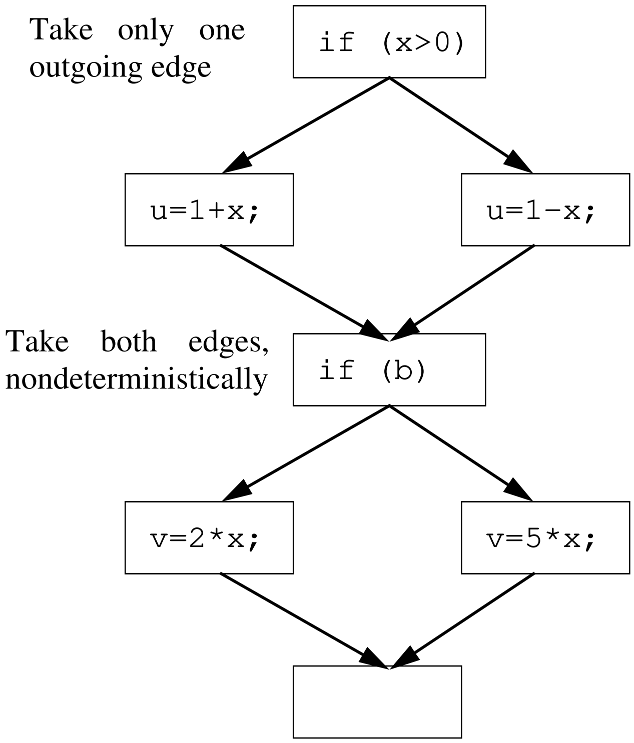
If there is an elegant solution to this problem, we still haven't found it. However, we did come up with an approach that is simple and easy to implement, with the disadvantage that the resulting setup code runs slowly.

The idea is to pretend that instead of executing the setup code on a normal computer, we were executing it on a non-deterministic computer. On such a computer you could decide, at a branch, to execute along both sides of the branch instead of just one of them. We further assume that this computer keeps track of code it has already executed, and never executes it again.

On this mythical machine its easy to produce setup code. The control flow for the setup code is the same as the control flow for the original code, with the exception that wherever we had a branch on a dynamic value in the original code, we replace it with one of these non-deterministic branches.

The result is that the setup code, during any single run of the program, executes precisely the set of static operations that are reachable in the original program, given the current values of the run-time constant branch predicates. This is exactly what we want.

You can see this in the example above, where the first branch is copied intact, since it depends on a static predicate, and the second branch is converted into a non-deterministic branch that executes both sides.



Of course we don't really have such a non-deterministic machine; instead we simulate one. The compiler generates the setup code by introducing new variables into the program to store the extra state needed simulate such a machine: an extra bit for each basic block to make sure setup code isn't executed twice, and a stack that stores basic blocks that the simulator intends to evaluate later after it finishes with whatever basic block it is currently evaluating. The original flow control is augmented by operations that push and pop basic blocks off this stack as necessary. This process is similar in character to the transformation used to convert a non-derministic Turing machine into a deterministic one.

In the end the setup code executes the proper static operations in the proper order, but it does so much more slowly than normal code. For each static operation the setup code executes, it has to do of extra work: checking bits to make sure it doesn't execute any setup code twice, pushing basic blocks on the stack at non-deterministic branches, and then popping the basic blocks off the stack later.

Additionally, the resulting setup code is difficult to optimize statically, because its control flow is encoded in the new simulation code, where the compiler can't take advantage of it. The compiler can't even register allocate variables in the setup code, for this reason. This problem is ameliorated by the fact that setup code is only run once, but it still has a noticable effect on performance. But at least it works.

3.2 Oracle

We have to do the run-time constant based optimizations dynamically because the values of run-time constants aren't known until during program execution. However, if we somehow magically could guess these values, then we could apply these optimizations at part of static compilation. In fact, we could apply better (but more expensive) versions of these algorithms because we have more time and more information offline then online. In practice of course the computer can't guess. However, if you run the same program twice on the same input, the values of the run-time constants will be the same both time. If you use the first run as a profiling run and save the run-time constants in a file, then you can then use them to optimize the program statically, so that when you run it again it runs with the optimum speedup. I implemented an extension to our system that just does this, and I call it the Oracle.

The implementation of the oracle was very simple because it leverages off of what we had already done. The BTA analysis is precisely the same, except instead of generating setup and template code, the oracle generates only template code, and instead of leaving holes in the template code, the oracle fills in run-time constant values from a file (unrolling loops where appropriate). The code then passes through the rest of the Multiflow pipeline where constant-oriented optimizations are applied as part of the standard compilation process--exactly as if the constants had been hard-coded in by the programmer.

The oracle is primarily useful as a research tool, because it allows us to compare our current dynamic compiler with a static compiler doing the same optimization. If we assume that the static compiler is doing a good job, this lets us see which of the optimizations we are doing well dynamically and which we are not. When a program that we think should be improved by dynamic compilation doesn't run any faster, we can use the oracle to determine whether the problem is that our dynamic compiler isn't optimizing the code well, or whether there just aren't as many opportunities for optimization as we thought.

4 Results

4.1 Accomplishments

This binding time analysis sucessfully provides the information we need to do dynamic compilation. Based on the test cases we have given it, it gets most of the run-time constants, especially the ones we are interested in, and never ever marks something as a run-time constant that isn't. It handles unstructured code gracefully. The time for binding time analysis was not a major part of compile time. The transformation based on the binding time information correctly separates code into setup and template. When the setup and template is compiled and fed into our dynamic compiler, the resulting dynamically optimized code shows significant speedups for a variety of programs. In short, it works.

4.2 Limitations of the Binding Time Analysis

While it was good enough for the first version of our dynamic compilation system, the binding time analysis still has significant limitations.

- The BTA only works on variables, so it can't deduce run-time constants in memory that are computed during setup code. For example if the code writes a run-time constant value to a memory location, and then reads it back, the BTA can't deduce that the result read back is still run-time constant.
- The BTA is intraprocedural, which means that run-time constants can neither be propaged into or out of functions called from within the dynamic region. This limitation can be overcome to some extent by inlining function calls, but only at the expense of an increase in code size, and not in cases of recursion or where the target function is variable.
- The BTA is limited in the kinds of loops it can handle: it only handles loops where after each iteration the next iteration is completely determined by run-time constants. It can't handle loops where after an iteration there are multiple possible next iterations (chosen based on dynamic conditions), and where the programmer wants to unroll all possibilities dynamically. It also can't handle loops based on recursion.
- The BTA can't handle the case where some value computed inside the region is annotated

as static. Currently all static annotations must occur on entry to the region.

We have ideas on how to fix these limitations, and hope to eliminate at least some of them in the next version of our system. There are also more vague limitation that are the basis of future research. It would be nice if:

- The BTA didn't need programmer annotations, but instead identified . This could be based on some combination of static analysis and profiling.
- The BTA could handle the concepts of slowly varying variables or variables that only change under certain conditions, and deduce useful information based on them.
- The BTA could deduce more general program properties that are "run-time constant," such as whether a scientific loop has data dependencies or whether an array bounds check is unecessary.

This list is of course endless.

5 Related Work

6 Bibliography