Accurate Static Branch Prediction by Value Range Propagation

Jason R. C. Patterson
(jasonp@fit.qut.edu.au)

School of Computing Science
Queensland University of Technology
Brisbane, Qld 4001, Australia
Phone: +61-75-316191

Abstract

The ability to predict at compile time the likelihood of a particular branch being taken provides valuable information for several optimizations, including global instruction scheduling, code layout, function inlining, interprocedural register allocation and many high level optimizations. Previous attempts at static branch prediction have either used simple heuristics, which can be quite inaccurate, or put the burden onto the programmer by using execution profiling data or source code hints.

This paper presents a new approach to static branch prediction called *value range propagation.* This method tracks the weighted value ranges of variables through a program, much like constant propagation. These value ranges may be either numeric or symbolic in nature. Branch prediction is then performed by simply consulting the value range of the appropriate variable. Heuristics are used as a fallback for cases where the value range of the variable cannot be determined statically. In the process, *value range propagation* subsumes both constant propagation and copy propagation.

Experimental results indicate that this approach produces significantly more accurate predictions than the best existing heuristic techniques. The *value range propagation* method can be implemented over any "factored" dataflow representation with a static single assignment property (such as SSA form or a dependence flow graph where the variables have been renamed to achieve single assignment). Experimental results indicate that the technique maintains the linear runtime behavior of constant propagation experienced in practice.

1 Introduction

The ability to predict at compile time the probability of a particular branch being taken provides valuable information for several optimizations, including global instruction scheduling, code layout, function inlining, interprocedural register allocation and many high level optimizations. For most of these optimizations branch predictions are merely an extra piece of useful information, but for global instruction scheduling and instruction cache optimizations the accuracy of the branch predictions can make or break the optimization.

This paper appeared in the Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, June 1995, pages 67-78. (La Jolla, San Diego)

Unfortunately, it is often difficult to predict which branches will and won't be taken through a particular program. This is not because conditional branches are unpredictable in nature, however. Indeed conditional branches show surprisingly predictable behavior. Obviously some branches will depend on the particular input being processed, but in most cases the branching behavior of a program is largely independent of its input. It has been observed that most conditional branches take one direction most of the time, irrespective of the particular input being processed and even the type of program [FisherFreudenberger92].

Up to now there have basically been three approaches to obtaining branch prediction data for use in optimizations:

- heuristics based on nesting and coding styles
- real execution profiling
- source code hints supplied by the programmer

Most heuristic approaches try to identify common programming practices. They are often based on dubious assumptions and are sometimes no better than random guesses. A simple example is the 90/50 rule, which predicts that backward branches are taken 90% of the time and forward branches 50%. This is obviously far too crude to base important decisions on. It is possible to predict loop controlling branches fairly accurately using simple nesting heuristics and loop analysis, but simple heuristics perform very poorly for the non-loop branches which dominate the dynamic branch count of many programs [Smith81, Wall91].

More sophisticated heuristic approaches base their decisions on the datatypes and type of comparison used in the branch and the code in the target basic block [BallLarus93, Wagner+94]. Although these approaches are much better than trivial heuristics, their accuracy is still only mediocre. These approaches can also be somewhat hit-and-miss, since programs which use programming styles that don't fit the heuristics result in very poor predictions.

Execution profiling is based on instrumenting a program with code to collect information such as basic block execution counts, which can then be used to determine branch probabilities [McFarlingHennessy86, BallLarus92]. The instrumented version of the program must be run to collect the profile data, so programmer intervention is required. The input used during these test runs can have an effect on the resulting profile.

As you would expect, execution profiling is much more accurate than heuristics, usually within a few percent of the common case runtime behavior for most branches [FisherFreudenberger92]. However, it is not perfect. Unfortunately researchers often use the same input for obtaining the profile and measuring the effectiveness of optimizations based on it. This gives an unrealistically good picture of the situation, since in practice a program is optimized based on a small number of profile runs, then run with many different input sets. This different input data causes the profile to be a less-than-perfect match to the input. Despite this, execution profiling is still easily the most accurate method for obtaining branch prediction data.

There is, of course, a serious weakness to execution profiling - it is too inconvenient for all but the most performance aware programmers to “bother doing”. This is unfortunate, because profiling can help identify higher level performance problems as well. Sadly it is the case that many programmers don’t even bother identifying performance critical routines, let alone tuning their code for better performance. Another problem with execution profiling is that it is not practical for some pieces of code, such as parts of an operating system kernel.

The final method of obtaining branch predictions is through source code hints. Approaches based on programmer supplied hints are both inconvenient and potentially inaccurate. They do, however, provide a way to handle certain difficult situations where neither compile time prediction nor execution profiling are adequate. Like the “register” declarations in the C language, such hints are indicative of a lack of suitable compiler technology more than anything else.

Clearly, it would be nice to be able to predict branching behavior as accurately as execution profiling does, but without requiring any programmer intervention. This paper describes a new approach to static branch prediction which moves towards this objective. The technique is called *value range propagation*. It uses the mechanism of constant propagation [WegmanZadeck91] and some of the ideas from range analysis [Harrison77].

Value range propagation extends standard constant propagation to track the weighted value ranges of variables in a program. These value ranges may be either numeric or symbolic in nature. Branch predictions are then made based on these value ranges. In the process, *value range propagation* subsumes both constant propagation and copy propagation.

2 Related Work

Constant propagation is a well-understood problem whose goal is to identify expressions that are constant for every possible execution of a program and can therefore be evaluated at compile time rather than runtime [Kildall73]. This is a simple forward dataflow problem - information about the “constantness” of expressions is simply propagated around the control flow graph until a fixed point is reached.

There is a strong similarity between constant propagation and our objective. Instead of finding the expressions which are constant, we want to determine the weighted range of values an expression can have during the execution of a program. Branches based on this variable can then be accurately predicted simply by examining this weighted value range.

The technique outlined in this paper uses the mechanism of constant propagation with conditional branches developed by Mark Wegman and Ken Zadeck [WegmanZadeck91]. Their approach uses the *static single assignment* (SSA) representation, which was originally developed as a dataflow representation geared towards propagating values through a program. It therefore provides an ideal platform for this kind of analysis. Using SSA form, constant propagation can be performed in essentially linear time.

In SSA form every variable has only one assignment, and therefore each use has a single definition. In essence, the def-use chains (called SSA edges) become a one-to-many rather than a many-to-many relationship. For a detailed description of the SSA representation see [Cytron+91]. The transformation into SSA form involves two steps. In the first step special assignment operations called *ø-functions* are inserted at joins in the program where alternative versions of a variable meet. These ø-functions occur at the entrance to basic blocks, and have the form:

$$a = \phi(b, c, \ldots)$$

The meaning of this is that if control reaches this block via the first in-edge, a is assigned the value of b, if via the second in-edge, a is assigned the value of c, and so on. In the second step the variables are renamed so that each assignment to a particular variable has a unique name. This achieves the “static single assignment” property - each variable now has a single definition point (either in a real assignment or in a ø-function).

Although this paper describes the *value range propagation* method using the SSA form and it’s associated terminology, it can be implemented using any “factored” dataflow representation with a static single assignment property. In fact, my actual implementation uses a dependence flow graph [JohnsonPingali93] where the variables have been renamed to achieve single assignment.

3 Algorithm Overview

3.1 Objective

The objective of any branch prediction algorithm is to determine the likelihood of taking each conditional branch in a program. Most heuristic methods simply mark each branch as either “taken” or “not-taken”. This is useful but not ideal. What we would really like to know is the *probability* of a branch being taken. This is what *value range propagation* produces - the out-edges of each conditional branch are marked with probabilities ranging from 0 (never taken) to 1 (always taken). Using probabilities rather than a simple yes or no answer lets us accurately assess a series of branches to determine the degree of speculation involved in moving a particular instruction (in global instruction scheduling), or the likelihood of executing a particular basic block (for cache optimizations).

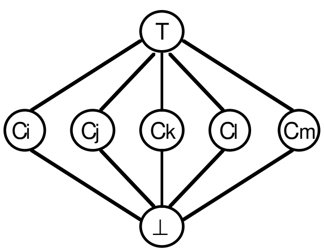
3.2 Propagation Terminology

The heart of the *value range propagation* method is the propagation of the weighted range of values each expression can have around the control flow graph until a fixed point is reached. In formal terms, the objective of any propagation algorithm is to determine an *output assignment* for each variable in a program. These output assignments represent some knowledge about the value of each variable, and are normally described in terms of a multi-level *lattice* of possible values. The highest level in this lattice is *top* (⊤), which represents an as yet undetermined value. Below this are the levels which represent some known property of the variable. Finally, the lowest level is *bottom* (⊥), which indicates that the variable contains a value that cannot be predicted at compile time.

Propagation algorithms start with the optimistic assignment of ⊤ for each variable, and proceed by lowering the assignments (in terms of the lattice) as more information is gathered about the program. Eventually a fixed point is reached and the process terminates. At this point no ⊤ assignments will remain, so each variable will either have some information assigned to it, or will be undefined (⊥). Information is gathered about particular variables by symbolically executing program statements and applying the meet operator (∩) at joins in the program (ø-functions). The meet operation must be defined such that the result is never higher than the operands, otherwise the algorithm might never terminate.

3.3 Value Range Propagation

In standard constant propagation there is only one middle level (*constant*), and the rules for expression evaluation and meet operations are very simple (Figure 1). If a variable has a constant as its output assignment after propagation, the variable will hold this constant value for all possible executions of the program. Although this is useful for constant folding and determining reachable code, this information is of no use for branch prediction.



any ∩ ⊤ = any

any ∩ ⊥ = ⊥

Ci ∩ Cj = Ci (if i = j)

Ci ∩ Cj = ⊥ (if i ≠ j)

Figure 1. The standard constant propagation lattice.

In order to predict what proportion of the time a branch will be taken, we really need to know what is in the variable that the branch is based on. Not only do we need to know the values that might be in the variable, but also how often each value actually occurs at runtime. Take a moment to consider the code segment below (Figure 2). For fun, work out how often block A is executed.

```
for (x=0; x<10; ++x)
{
    if (x >7) { y = 1; }
    else { y = x; }
    ...
    if (y == 1) { ... Block A ... }
    ...
}
```

Figure 2. A simple example.

To the human reader it is obvious that block A is executed 30% of the time. Subconsciously you probably used the following logic in coming to this decision:

- x is evenly distributed across the values 0-9
- the test *x>7* will succeed 20% of the time
- if the test succeeds *y* will have the value 1
- if the test fails *y* has a 1-in-8 (12.5%) chance of having the value 1
- after the test *y* has a (0.2*1 + 0.8*0.125) = 30% chance of having the value 1

The *value range propagation* method uses exactly this kind of logic to determine branch probabilities. As with standard constant propagation, a simple worklist algorithm is used as the mechanism for forward dataflow analysis. Two working lists are maintained: the *FlowWorkList*, which is a worklist of control flow graph edges, and the *SSAWorkList*, which is a worklist of SSA edges (def-use chains).

The major differences between *value range propagation* and standard constant propagation are:

- value ranges are propagated rather than constants
- the rules for expression and ø-function evaluation are somewhat more complicated
- loop carried expressions are detected and handled specially (so that we don't actually execute the loops!)
- each control flow graph edge has a probability associated with it rather than an *is-executed* flag

The algorithm proceeds as follows:

1 Initialize the FlowWorkList to contain the out-edges from the start node, the SSAWorkList to be empty, all probabilities to be 0, and all ranges to be ⊤. Mark the out-edges of the start node with a probability of 1 (ie: 100%).

2 If both working lists are empty, terminate. Otherwise extract the next item from one of the lists. Execution can proceed by processing items from either list, but experience has shown that preferring to select from the FlowWorkList tends to cause information to be gathered more quickly and therefore reduces the running time of the algorithm.

3 If the item is a control flow graph edge, visit the target node. If this node has never been visited before, evaluate every expression in the node, otherwise evaluate only the ø-functions. For each expression, if the new result (value range) differs from the old result for that expression (SSA variable), add the SSA edges starting at that expression to the SSAWorkList.

4 If the item is a ø-function, and one or more of the node's in-edges are back edges (as identified by a depth first traversal from the start node), then there is a loop in the SSA edge chain starting at this ø-function. In other words, this is a *loop carried* variable. Attempt to *derive* its value range. Loop carried variable derivation is described later in this paper. If derivation succeeds, mark the expression as derived (derived expressions are not to be re-evaluated). If derivation fails,

mark it as impossible to derive, so that derivation will not be re-invoked for it.

5 If the item is a ø-function which has not been marked as derived, and the probability of executing any of the in-edges to this node is non-zero, evaluate that ø-function. The evaluation of a ø-function is simply the merging of the appropriate ranges according to the current branch probabilities for each in-edge[1]. If the new result differs from the old result for that ø-function, add the SSA edges starting at that ø-function to the SSAWorkList.

6 If the item is an expression which has not been marked as derived, and the probability of executing any of the in-edges to this node is non-zero, evaluate the expression. If the new result differs from the old result for that expression, add the SSA edges starting at that expression to the SSAWorkList.

7 If the item is a conditional branch instruction which has not been marked as derived, then determine the probability of taking the branch by examining the relevant variable's value range. If the new probability differs from the old probability for this branch, mark the out-edges with the new probabilities and add any changed out-edges to the FlowWorkList.

8 Goto 2.

3.4 Range Representation

Choosing a flexible but efficient representation for the value range of a variable is a difficult but critical part of implementing the *value range propagation* algorithm. An ideal representation would provide a space efficient encoding for common cases such as single values, evenly distributed ranges and arithmetic sequences, whilst simultaneously allowing the flexibility to represent less common ranges such as geometric sequences and unusual sequences such as prime numbers. To make things even more complex, many ranges can only be specified relative to others, eg: *"x* is greater than $y+2$*"*. Such symbolic ranges must also be handled efficiently.

At one extreme, it is possible to use a completely general symbolic representation for range information. This may be appropriate for formal program verification systems, but it would be far too inefficient for practical use in a compiler. Using too simplistic a representation, on the other hand, yields results not much better than constant propagation. Obviously a tradeoff between accuracy and efficiency is necessary.

From a practical point of view, value range analysis is probably of most use in tracking the arithmetic sequences of loop control variables, dense ranges, and constants of various types. In the method described by this paper, the value contained by a particular variable is represented by a set of weighted ranges where the total range is the union of all the individual ranges. Each individual range is defined by:

- a probability
- a lower bound
- an upper bound
- a stride (arithmetic step size)

An even distribution is assumed within each range, so uneven distributions must be represented by multiple ranges. The notation used in this paper to describe a set of ranges is as follows:

{ P[L:U:S], ... }

In this notation *P* represents the probability of the particular range applying at runtime, *L* and *U* give the lower and upper bounds of the range, and *S* is the stride.

In addition to ranges between numeric lower and upper bounds with numeric strides, it is also necessary to handle common cases of symbolic ranges to achieve high levels of accuracy in branch predictions. This can be achieved by simply allowing each "number" in a range definition to be defined as:

SSA Variable   operator   Constant

For numeric values the variable component is NULL[2], and for purely symbolic values the constant component is +0. This representation allows the simple case of values defined relative to a single variable to be handled (eg: $x+2$), but not values which represent potentially complex operations between multiple variables (eg: $x+y$). As a result, range operations and comparisons are kept simple. Although this only allows meaningful operations and comparisons between variables which share a single common "ancestor", it achieves most of the practical benefits of symbolic analysis without having to implement a full-blown theorem tester to determine relationships between ranges.

As mentioned above, each variable is represented by a *set* of ranges. In order to eliminate the possibility of a single variable's set of ranges growing indefinitely, it is necessary to place an upper limit on the number of ranges used. In other words, a "give-up" point needs to be identified. In practice a relatively small number of ranges is adequate, normally no more than four for programs with "normal" control flow. Using four ranges per variable allows us to handle merges from up to two levels of conditional branching without losing accuracy.

3.5 Range Operations

The method described in this paper propagates information about the value ranges of variables through a program in the same way that constants are propagated in traditional constant propagation. As a result, it is necessary to extend the expression evaluation mechanism of constant propagation to handle value ranges. For example, the optimizer must be able to evaluate expressions such as:

```
    { 0.7[32:256:1], 0.3[3:21:3] }
+   { 0.6[16:100:4], 0.4[8:8:0] }
=   { 0.42[48:356:1], 0.28[40:264:1]
      0.18[19:121:1], 0.12[11:29:3] }
```

Implementing these range operations is relatively straightforward. Efficiency is important here because unlike constant propagation, expressions may need to be re-evaluated many times before a fixed point is reached. If expressions become very large, applying a technique such as Reif and Lewis's tree representations

[1] The probability of an in-edge from an unconditional branch (or fall through) is the sum of the probabilities of the in-edges of the first node with equivalent control dependencies to the node which ends with the unconditional branch (ie: the earliest node which dominates it and which it postdominates). In other words, the sum of the probabilities of the edges which lead to the node being executed.

[2] By NULL I mean whatever the null concept for the representation being used is. In my implementation, for example, it is virtual register 0.

[ReifLewis77] may be useful to avoid redundant re-computations. Unfortunately this technique implies a significant memory overhead, so practical uses are limited to very high-cost expressions.

The observant reader might have noticed that the above example actually *loses* information. Consider the calculation of 0.3[3:21:3] + 0.6[16:100:4]. The result shown above is 0.18[19:121:1], which represents any number from 19 to 121. In reality, however, there is no combination of a number from the range 0.3[3:21:3] and a number from the range 0.6[16:100:4] that will total 20. This indicates a limitation of our range representation - achieving more accuracy in this example would require either a more sophisticated range representation or a very large number of very small ranges. Although this loss of accuracy leads to less accurate branch predictions, providing additional accuracy would greatly increase the running time of the algorithm.

Some range operations are problematic for other reasons. Load operations, for example, typically result in a range of $\perp$ unless detailed alias analysis information is available. As a result, conditional branches based on a value loaded from memory often cannot be predicted using *value range propagation*. Instead, heuristics similar to those in [BallLarus93] must be used. Depending on the quality of the alias analysis being performed and the type of program being optimized, this might occur anywhere from 10% to 90% of the time. To determine probabilities rather than simple "taken" or "not taken" predictions the heuristics must be weighted and combined using a technique such as [WuLarus94].

3.6 Loop Carried Expressions

The standard constant propagation algorithm propagates values without explicitly handling loops in the program. For constant propagation this is acceptable because an expression can only be evaluated twice. For example, a loop control variable is first assigned the constant of its initial value, then when the incremented version of this value is propagated around from the bottom of the loop the values are different so the variable is assigned $\perp$.

If we used this approach for *value range propagation*, each loop would "execute" as many times during propagation as it would at runtime, because the value range of the loop control variable would gradually be expanded until the loop termination test succeeded. Although this would work correctly, it would make *value range propagation* far too slow for practical use. To avoid this, it is necessary to explicitly identify loop carried expressions and determine their value ranges using a different approach.

Loop carried SSA variables are those with loops in their SSA edge chain. That is, one or more of the in-edges to a ø-function is a back edge (as identified by a depth first traversal from the start node). This is easily identified during *value range propagation* (it was step 4 in the algorithm description). Having identified a loop carried expression, it is desirable to determine its value range without having to "execute" the loop. This can be done by examining the expression's *derivation* [Harrison77].

A variable's *derivation* is the set of operations performed on that variable during the loop. In its most general form, a derivation would be a set of symbolic equations which specify how the variable may be computed from its previous value and the values of other variables. For practical purposes, however, this would be too complicated and would gain little over a simpler approach. In practice, it is only necessary to identify simple inductive derivations such as loop counters. This can be done by simply processing the operations in the derivation and matching them to a template such as:

new value = old value + set of possible increments
assert (new value between specific bounds)

The first part of the template matches the loop increment operation(s), and the second part matches the termination test(s). This template can then be combined with the initial value of the variable to produce a final value range.

If the set of operations in a derivation cannot be matched to such a template, then we can simply allow the propagation algorithm to determine the value range by "executing" the loop. Thus one should view derivation matching as an efficiency optimization. Only the simple derivation processing mentioned above is actually essential for reasonable performance, but adding more templates and more powerful derivation processing reduces the need for brute force propagation to do the work.

3.7 Interprocedural Analysis

Standard constant propagation is often considered difficult to extend to interprocedural cases because solving the fully general case where constants may be propagated through intermediate procedures requires symbolic information to be manipulated rather than simple constants. In contrast, since we are already performing relatively sophisticated symbolic analysis for intra-procedural *value range propagation*, it is straightforward to extend our analysis to handle interprocedural cases as well. In fact a significant amount of branch prediction accuracy would be lost if we did not handle interprocedural cases, because the branching behavior of many procedures depends greatly on the values of their parameters.

Interprocedural constant propagation is usually described in terms of a set of *jump functions* associated with each call site [Callahan+86, GroveTorczon93]. These jump functions give the values of the actual parameters used in each call of a function, and the value of each formal parameter is simply found by merging the values of the jump functions at the various call sites[3]. In our case, the jump functions map directly to the range representations for the parameters in the call, and the propagation algorithm remains the same. In essence, the entire program is treated almost as if it were one huge control flow graph.

One particularly important extension of interprocedural *value range propagation* is the judicious use of *procedure cloning* for critical procedures [CooperHallKennedy92]. Procedure cloning involves duplicating a critical procedure which is not inlined but which is called in two (or more) significantly different contexts so that each copy may be optimized in a different way. Thus the distinct copies can take advantage of their specific calling contexts. Since the calling context has a large impact on the branching behavior, this leads to substantially more accurate predictions. Many other optimizations may also benefit from procedure cloning [MetzgerStroud93].

3 Procedure return values are handled similarly, using *return jump functions*.

3.8 *Putting It All Together*

As a summary, consider our earlier example program (Figure 2). The control flow graph in SSA form is shown below complete with SSA edges (Figure 3).

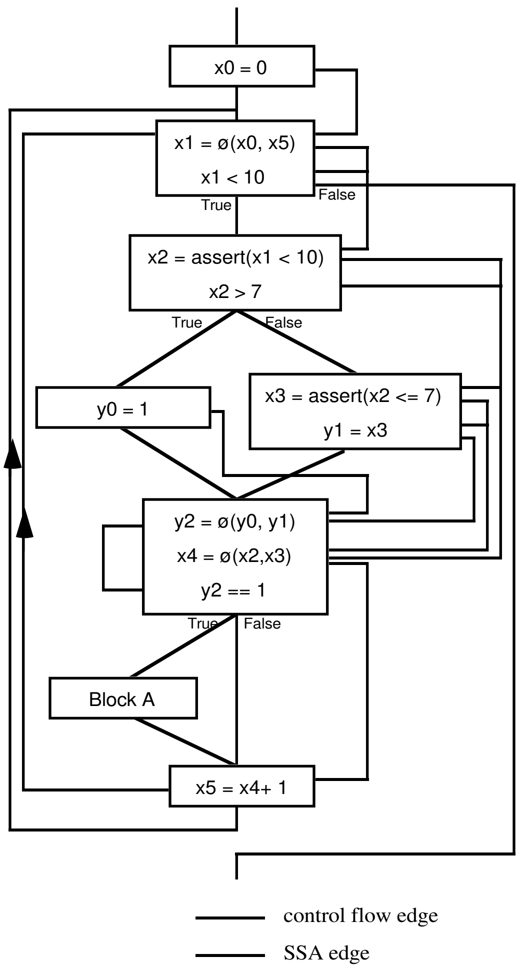


Figure 3. Our simple example in SSA form.

Notice the assertion along the true edge of the x1 < 10 branch. Valuable information can often be derived from the equality tests controlling branches, so assertions such as this one are placed in the graph after conditional branches to assert specific properties of a variable[4].

Now consider the application of the *value range propagation* algorithm. Initially, all branches are marked with probabilities of 0 and all variables are assigned $\top$. The starting edge is marked with 100% and selected for processing. This causes the evaluation of all the expressions in the first block, which includes both the assignment to x0 *and* the unconditional branch (or fall through). The assignment to x0 yields a range of 1[0:0:0]. Since this is not the same as $\top$, the SSA edge starting at this expression is added to the SSAWorkList. The unconditional branch (or fall through) is marked with a 100% probability, and is added to the FlowWorkList.

The out-edge of the first block, or perhaps the SSA edge starting at x0, is selected next, causing the ø-function assignment to x1 to be evaluated. One of the in-edges to this ø-function is a back edge, so we attempt to *derive* the ø-function's value range. In this case derivation will succeed and result in a range of 1[0:10:1]. Again this is different from $\top$, so the two SSA edges starting at this expression are added to the SSAWorkList.

The branch on the expression x1 < 10 is evaluated next. Since x1 has the range 1[0:10:1], the branch is predicted as 91% taken. As a result, the true out-edge is marked with a 91% probability and the false edge a 9% probability. Since the probabilities of both edges have changed, both edges are added to the FlowWorkList.

This process continues until the FlowWorkList becomes empty. At this stage the SSAWorkList still has many entries however, and the analysis is far from complete. In fact this typically represents the point at which each expression has only been evaluated once. One by one, each SSA edge is now removed from the SSAWorkList, and each in turn causes further information to be propagated around the graph. Although our simple example terminates rather quickly, in general new edges may be added to either or both worklists, and the value ranges and branch probabilities may be further refined.

Eventually these re-evaluations begin to produce the same results as the results previously calculated. In these cases no edges are added to the working lists, so the lists gradually become shorter. The process slowly winds down and the FlowWorkList and SSAWorkList become empty. At this stage a fixed point has finally been reached. The final value ranges and branch probabilities for our simple example are shown in Figure 4.

Value Ranges

x0	{ 1[0:0:0] }
x1	{ 1[0:10:1] }
x2	{ 1[0:9:1] }
x3	{ 1[0:7:1] }
x4	{ 1[0:9:1] }
x5	{ 1[1:10:1] }
y0	{ 1[1:1:0] }
y1	{ 1[0:7:1] }
y2	{ 0.8[0:7:1], 0.2[1:1:0] }

Branch Probabilities

x1 < 10	91%
x2 > 7	20%
y2 == 1	30%

Figure 4. Results for our simple example.

4 Algorithm Efficiency

Standard constant propagation can only lower (in the lattice sense) a variable twice, first to a constant and then to $\perp$. The asymptotic time complexity of standard constant propagation is therefore O(E.V). This is a worst case result, suggesting that every variable is defined in every basic block. Both intuition and empirical

4 Variables created by assertion are a special case when merged in ø-functions. The merging of an assertion-derived variable with it's "parent" variable, or of all the assertion-derived variables of a common parent, results in the value range of the parent variable.

evidence suggest that in practice it is closer to O(E) (ie: linear in the size of the program).

The method proposed in this paper adds a significant amount of processing to that of standard constant propagation. Unlike constant propagation, the lattice for *value range propagation* has a conceptually infinite number of middle levels, and expressions inside loops may need to be re-evaluated many times before a fixed point is reached. The expression evaluation itself is also more complex than in constant propagation - up to R^2 operations are performed per expression evaluation, where R is the maximum number of ranges used to represent a variable (normally four).

In order to guarantee termination, it is necessary to show that each variable can only be lowered a finite number of times. Placing an upper bound on the number of value ranges constituting a variable's representation has exactly this effect, but using this as a proof of termination is somewhat unsettling since it merely guarantees that any particular loop might only iterate several million times!

A more realistic estimate of the algorithm's efficiency can be provided by noting three observations:

- Non-loop dependent ranges are only evaluated once.
- Almost all loop dependent ranges are either derived or are completely narrowed after a single evaluation because the variables they depend on are derived.
- Many problematic ranges cannot be represented and quickly become $\perp$. For example, using the restricted representation described in this paper even a geometric sequence cannot be represented. In these cases the propagation algorithm will attempt to enumerate all the values in the range. Due to the finite nature of the range representation, this leads to either an overly broad representation (as was the case in our earlier range operation example), or $\perp$. In either case no further refinement is possible.

Noting the above observations, it is reasonable to expect *value range propagation* to exhibit similar runtime behavior to that of constant propagation. This is indeed the case - *value range propagation* is slower than constant propagation, but still linear in the size of the program. Figures 5 and 6 show the runtime behavior for a collection of 50 programs including the SPEC92 suite, various other benchmarks and numerous UNIX utilities (evaluation sub-operations take essentially constant time).

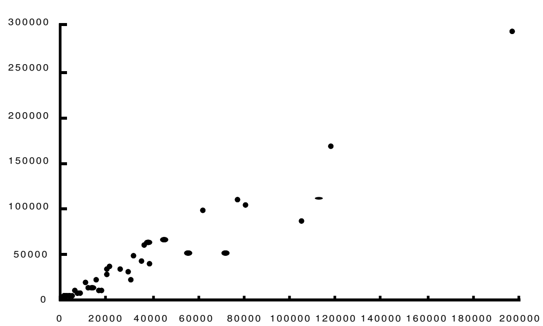


Figure 5. Number of expression evaluations versus number of instructions.

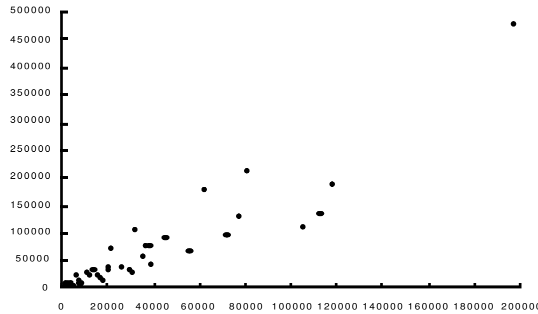


Figure 6. Number of expression evaluation sub-operations versus number of instructions.

5 Results

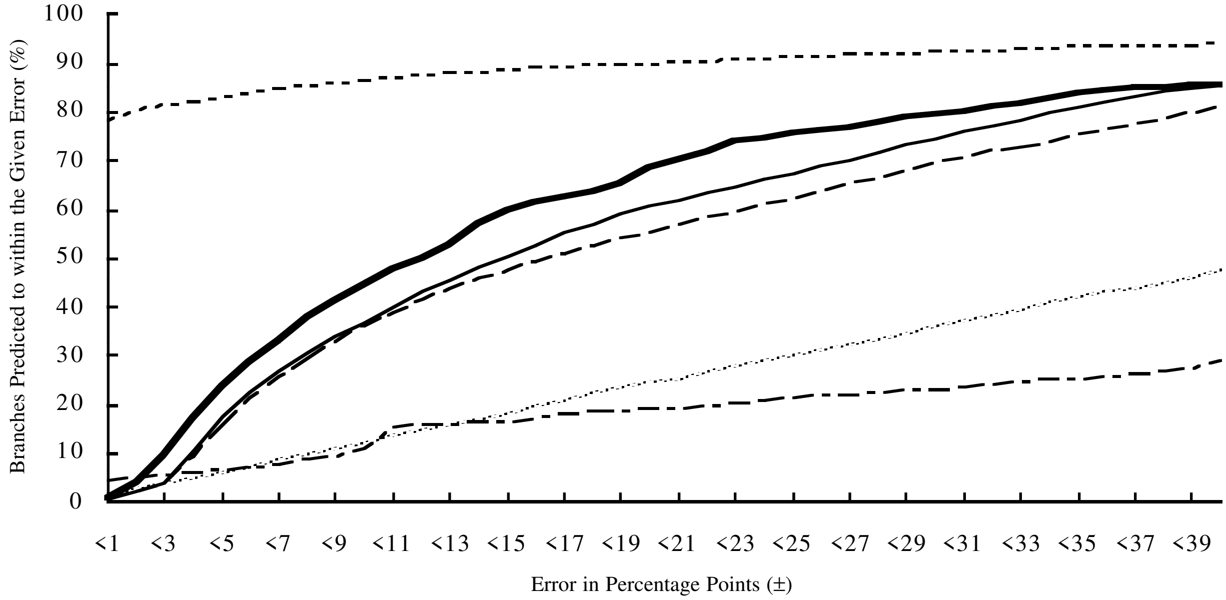
To determine the accuracy of the *value range propagation* approach for real programs, branch predictions determined using *value range propagation* were compared to the results of execution profiling and also to the results achieved using simple and complex heuristics (the 90/50 rule and the [BallLarus93] heuristics combined as in [WuLarus94] to produce probabilities). Different inputs were used to collect the execution profiles and the "actual observed behavior", reflecting the normal use of execution profiles found in practice. Heuristics similar to those of [BallLarus93] were used in cases where the *value range propagation* algorithm encountered a branch with a variable whose value range was $\perp$. The SPEC92 benchmarks were used for the analysis.

The resulting branch predictions were analyzed in terms of how far each branch's predicted probability deviated from its actual behavior. This involved determining the difference between the predicted probability for each branch and the actual probability observed for that branch when the program was given the SPEC reference inputs. The analysis was done in both an unweighted context, where each branch contributed equally, and in a context where each branch was weighted according to its execution count.

This analysis is somewhat more detailed than that of other branch prediction studies, which normally consider only taken or not-taken results. That is, a branch is predicted as taken or not-taken, and if at runtime it is taken (or not-taken) more than half the time, the prediction is considered correct. In our case we are interested in determining how accurate our predictions are at the finer level of branch probabilities, not simply branch directions. Since we will be speculating on the usefulness of scheduling instructions based on these predictions, it is important to know exactly how close to the real behavior our predictions are. Consider, for example, the decision of whether to speculatively move an instruction up through two conditional branches. If each branch is taken 60% of the time, our instruction will only be useful 36% of the time. If the branches are simply predicted as taken, however, you could be led to believe that the instruction would be useful most of the time.

Figures 7 and 8 show the weighted and unweighted results for the SPECint92 and SPECfp92 benchmark suites. Each benchmark is weighted equally within its suite. The graphs plot the percentage of branches predicted to within a given error margin.

SPECint92 Unweighted



SPECint92 Weighted

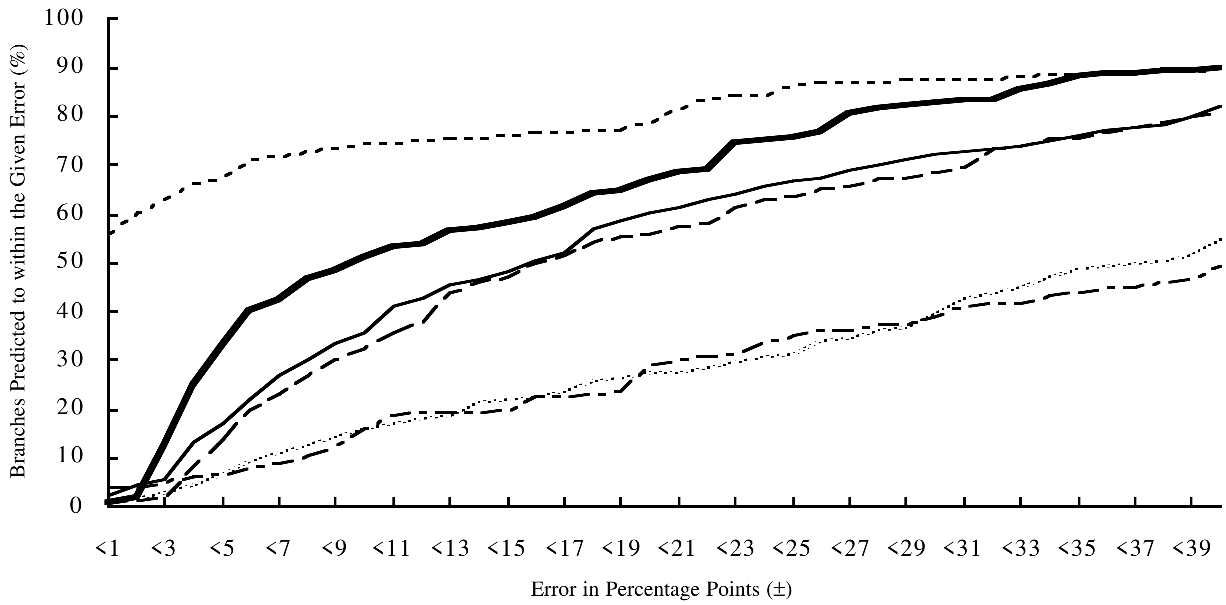
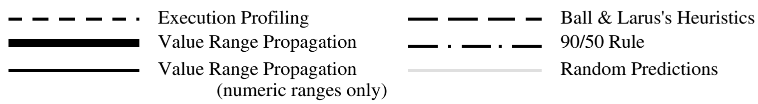
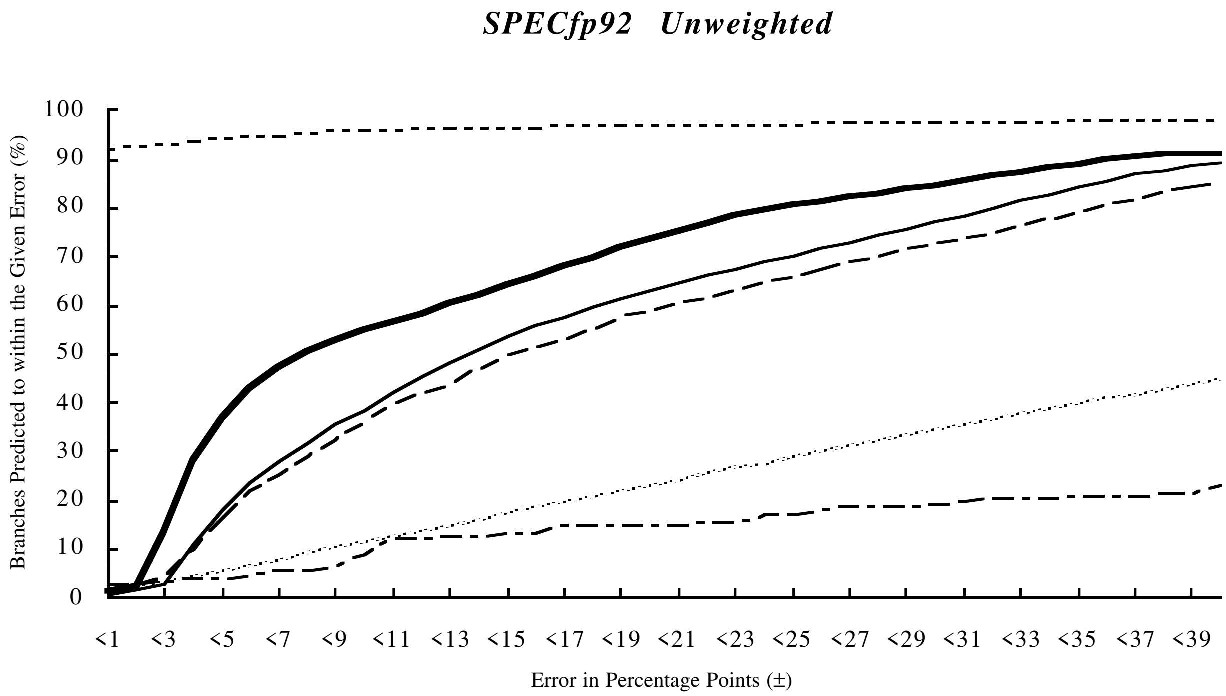


Figure 7. Results for SPECint92 (unweighted & weighted by execution count).

Legend





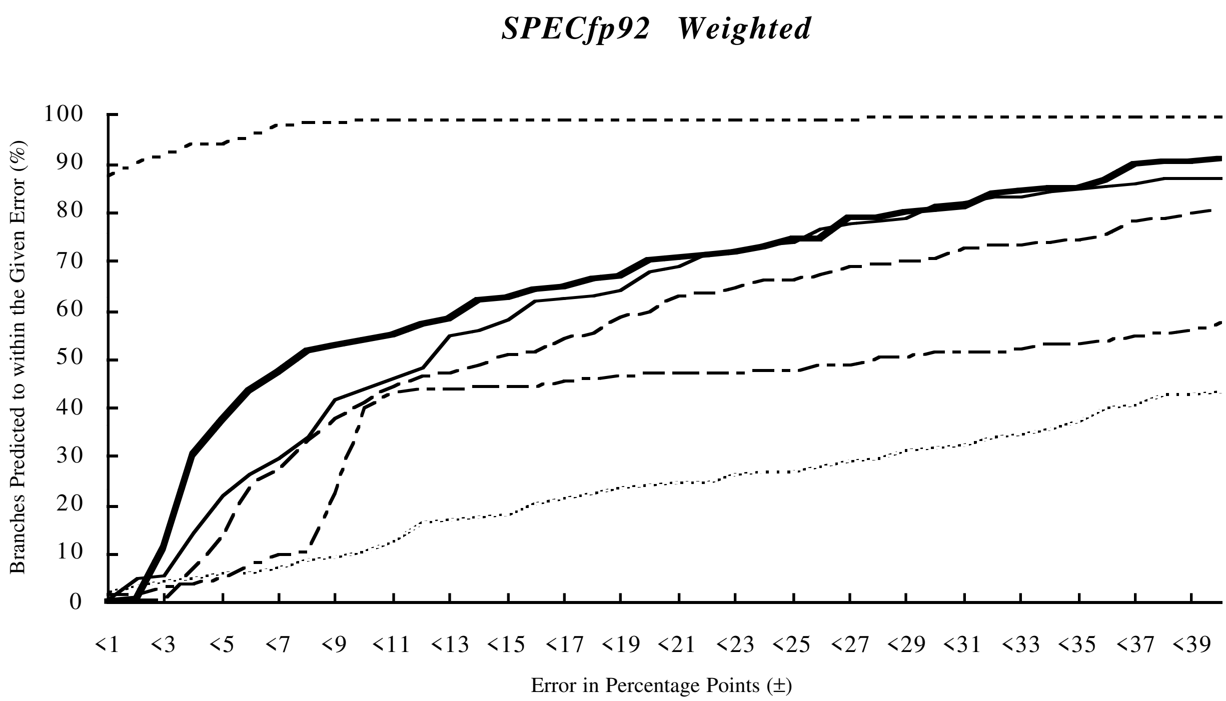


Figure 8. Results for SPECfp92 (unweighted & weighted by execution count).

Legend

Execution Profiling
Value Range Propagation
Value Range Propagation (numeric ranges only)
Ball & Larus's Heuristics
90/50 Rule
Random Predictions

The error margin plotted is between 0 and 40 percentage points. This range was chosen to accentuate the important part of the graph (the low error margins). Techniques which cannot predict a substantial percentage of branches to within ±40% are doing very poorly indeed, so displaying the remaining 40 - 100 error range would add little meaning to the graphs.

Since the graphs plot the percentage of branches predicted to within a given error margin, better predictions result in lines which move more quickly towards the top of the graph. For reference, the perfect static predictor would mark each branch with the same probability as was observed in the trial runs, and would have a graph with a horizontal line across the top, indicating that 100% of the branches were predicted to within ±0% of the observed behavior.

As you would expect, execution profiling does extremely well in all cases. It does almost as well as the perfect static predictor for numeric code (SPECfp92), with an astonishing 92% of branches predicted to within ±1% of their actual behavior! For integer and pointer intensive code the results are slightly worse but still very impressive (SPECint92). Interestingly, the weighted integer results for execution profiling are significantly worse than the unweighted results. This may be because many of the branches which get heavily weighted in SPECint92 are controlled by the size of the input, and the SPEC feedback collection inputs (input.short) are much shorter than the reference inputs (input.ref).

As the figures show, the predictions based on *value range propagation* are quite accurate. Results are shown for the *value range propagation* algorithm both with and without symbolic ranges. In both cases they are better than the heuristic methods. Adding symbolic ranges substantially increases the overall accuracy of the method because many more branches can be predicted without resorting to heuristics. Whether adding even more sophisticated symbolic analysis would result in a significant increase in accuracy or not is an open question. Unfortunately, even a small increase in the scope of the symbolic range representation would require a large amount of implementation effort. Interestingly, the *value range propagation* method is significantly more accurate for numeric code than for integer and pointer code. This is largely because numeric code often has a very simple branching structure, with most branches depending on loop control variables. This reduces the number of cases where branches are based on variables whose value range is $\perp$, which in turn reduces the amount of time when heuristics are used.

As expected, the 90/50 rule does most poorly, particularly in the unweighted case. This is largely due to the inaccuracy of the "50" part of the 90/50 rule. Backward branches *are* usually taken, so 90% is often quite close to the actual behavior of these branches, but forward branches depend greatly on the particular test being performed. Most branches take one direction with high probability, and very few forward branches are a 50/50 split. As a result, the graph for the 90/50 rule has a sudden increase at about the 50% error mark (not shown in the figures), since all the branches which the 90/50 rule predicts as 50% taken are suddenly accounted for (50% ± 50% covers the entire range of possibilities!).

6 Applications

One of the most pleasing features of the approach outlined in this paper is that for those who already compute the SSA form or a similar representation, this approach can be added to an optimizer as an extension of constant propagation. So with relatively little effort, quite accurate branch predictions are suddenly available. These can then be used in a host of optimizations. The three main optimizations which benefit from branch predictions are:

Global Instruction Scheduling

> Superscalar and VLIW processors issue several instructions every cycle. To fully utilize such processors, compilers must find several independent instructions to issue for each cycle, or otherwise let the processor sit partially idle. Due to the relatively small size of most basic blocks, it is usually necessary to look beyond basic block boundaries and perform some form of global scheduling with speculative execution [BernsteinRodeh91] or trace scheduling [Fisher81, Ellis85]. Good branch predictions are of great benefit here since the degree of speculation involved in moving a particular instruction can be accurately assessed.

Code Layout, Cache Optimizations & Inlining

> On many modern processors, an instruction cache miss or pipeline flush due to an unexpected change of direction in the instruction stream will cost several cycles. Even a correctly predicted branch may cost more than straight line code. As a result, compilers must pay careful attention to the way they lay out their generated code. This usually means placing related pieces of code close to each other, inlining simple function calls and coding likely paths as straight-line code with branches to less likely code which is placed out-of-line [McFarling89, PettisHansen90]. The objective here is to reduce the number of branches encountered at runtime and to improve the I-cache hit rate so that fewer misses occur. This approach can consistently make an I-cache appear 2 or 3 times as large as it does under current practice.

Interprocedural Register Allocation

> Modern processors typically have large register sets which, if used well, can dramatically reduce the memory reference overhead. Unfortunately, the frequent occurrence of function calls in most programs significantly reduces the effectiveness of the register set. Register windows are a hardware solution to this problem, but interprocedural register allocation which takes into account the probabilities of function calls can make much better use of a given register set [Wall86, Wall88, Wall91, SteenkisteHennessy89].

In addition to the three optimizations mentioned above, several traditional high level optimizations can also benefit from knowledge of frequently executed paths by using tail duplication to create what are effectively larger basic block structures [ChangMahlkeHwu91]. A similar mechanism can also be used to improve the accuracy of branch prediction itself [Krall94].

Branch probabilities can also be used to control the order of applying other optimization phases, as is done in *coagulation* [Karr84, Morris91]. In this case what we want to know is the execution frequencies of functions and basic blocks, not the probabilities of branches. This information can be obtained by using a Markov state transition model [Wagner+94], or by propagating frequencies around the control flow graph until a fixed point is reached [WuLarus94]. Optimizations can then be applied in descending order of execution frequency. This is particularly effective for optimizations which allocate a limited resource, such as register allocation, since the most frequently executed code is processed first and is therefore more likely to get the resources it needs.

In addition to using the branch probabilities calculated from *value range propagation*, the value ranges themselves can also be used in some simple array access optimizations. Of particular note are:

Alias Analysis for Array Accesses

Using *value range propagation* it is sometimes possible to show that the ranges of the indices of two array accesses cannot overlap. As a result, these two accesses cannot alias each other. This analysis is much more limited than sophisticated data dependency analysis techniques such as Banerjee's Inequalities [Banerjee88]. However it does offer a simple false-dependency breaking mechanism for compilers which don't implement the more sophisticated methods.

Elimination of Array Bounds Checks

For languages which require (or compilers which implement) dynamic array bounds checking, many array bounds checks can be shown to be redundant by *value range propagation*. The elimination of these tests in combination with existing test minimization techniques such as [Gupta93] can greatly reduce the overhead of array bounds checking.

Finally, *value range propagation* itself can be viewed as an optimization. Viewed in this context, *value range propagation* subsumes both constant propagation and copy propagation. If a variable's final value range is a single constant such as 1[7:7:0], then the variable's value is constant for all possible executions of the program and can therefore be evaluated at compile time. Similarly, a variable x whose value range is the single symbolic range of another variable such as $1[y:y:0]$ is simply a copy of y. As such, all references to x can be replaced by references to y, and x can be eliminated. Just as constant and copy propagation identify unreachable code, so does *value range propagation* - branches to unreachable code have a probability of 0.

7 Conclusion

This paper presented a new approach to accurate static branch prediction called *value range propagation.* The technique tracks the weighted value ranges of variables through a program, much like constant propagation. These value ranges may be either numeric or symbolic in nature. Branch prediction is then performed by simply consulting the value range of the appropriate variable. In the process, *value range propagation* subsumes both constant propagation and copy propagation.

Both numeric and symbolic ranges must be handled to achieve high levels of accuracy in branch predictions, and a tradeoff between accuracy and efficiency is required. For practical purposes a simple range representation capable of handling arithmetic sequences is sufficient, and a set of four ranges per variable is adequate for most programs with typical control flow. A representation which handles symbolic ranges defined relative to a single variable achieves most of the benefits of symbolic analysis at a reasonable cost in implementation effort. Whether more sophisticated symbolic analysis would result in more accurate predictions or not is an open question.

To avoid "executing" loops in the program, loop carried expressions must be detected and handled specially. This can be done by matching a loop carried variable's *derivation* to a set of templates which identify common looping scenarios. Variables whose derivations don't match a template can still be handled by allowing the propagation algorithm to "execute" the loop.

Experimental results indicate that the *value range propagation* approach gives predictions which are substantially more accurate than the best current heuristic approaches. Heuristics must still be used in the *value range propagation* method for branches with variables whose ranges are impossible to determine statically, such as loads from memory.

The *value range propagation* method can be implemented over any "factored" dataflow representation with a static single assignment property (such as SSA form or a dependence flow graph where the variables have been renamed to achieve single assignment). The method can be implemented as an extension of standard constant propagation for those who already compute the SSA form or a similar representation. It maintains the linear runtime behavior of standard constant propagation experienced in practice.

Acknowledgments

I would like to thank my PhD supervisor, Dr. John Gough, for providing many helpful comments during this work and for reviewing various drafts of this paper.

References

[BallLarus92]
Thomas Ball and James R. Larus. *Optimally Profiling and Tracing Programs*. Proceedings of the 19th Annual Symposium on Principles of Programming Languages, January 1992, pages 59-70.

[BallLarus93]
Thomas Ball and James R. Larus. *Branch Prediction For Free.* Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation, June 1993, pages 300-313.

[Banerjee88]
Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988.

[BernsteinRodeh91]
David Bernstein and Michael Rodeh. *Global Instruction Scheduling for Superscalar Machines.* Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation, June 1991, pages 241-255.

[Callahan+86]
David Callahan, Keith D. Cooper, Ken Kennedy and Linda Torczon. *Interprocedural Constant Propagation.* Proceedings of the SIGPLAN '86 Conference on Compiler Construction, June 1986, pages 152-161.

[ChangMahlkeHwu91]
Pohua P. Chang, Scott A. Mahlke and Wen-Mei W. Hwu. *Using Profile Information to Assist Classic Code Optimizations.* Software Practice and Experience 21(12), December 1991, pages 1301-1321.

[CooperHallKennedy92]
Keith D. Cooper, Mary W. Hall and Ken Kennedy. *Procedure Cloning.* IEEE 1992 International Conference on Computer Languages, April 1992, pages 96-105.

[Cytron+91]
Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman and F. Kenneth Zadeck. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph.* ACM Transactions on Programming Languages and Systems 13(4), October 1991, pages 451-490.

[Ellis85]
John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* PhD Thesis, Yale University, February 1985. Also available from MIT Press, 1986.

[Fisher81]
Joseph A. Fisher. *Trace Scheduling: A Technique for Global Microcode Compaction.* IEEE Transactions on Computers 30(7), July 1981, pages 478-490.

[FisherFreudenberger92]
Joseph A. Fisher and Stefan M. Freudenberger. *Predicting Conditional Branch Directions From Previous Runs of a Program.* Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1992, pages 85-95.

[GroveTorczon93]
Dan Grove and Linda Torczon. *Interprocedural Constant Propagation: A Study of Jump Function Implementations.* Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation, June 1993, pages 90-99.

[Gupta93]
Rajiv Gupta. *Optimizing Array Bound Checks Using Flow Analysis.* ACM Letters on Programming Languages and Systems 2(1-4), March-December 1993, pages 135-150.

[Harrison77]
William H. Harrison. *Compiler Analysis of the Value Ranges for Variables.* IEEE Transactions on Software Engineering 3(3), May 1977, pages 243-250.

[JohnsonPingali93]
Richard Johnson and Kershav Pingali. *Dependence-Based Program Analysis.* Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation, June 1993, pages 78-89.

[Karr84]
Michael Karr. *Code Generation by Coagulation.* Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, June 1984, pages 1-12.

[Kildall73]
G. A. Kildall. *A Unified Approach to Global Program Optimization.* Proceedings of the First Annual Symposium on Principles of Programming Languages, October 1973, pages 194-206.

[Krall94]
Andreas Krall. *Improving Semi-Static Branch Prediction by Code Replication.* Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation, June 1994, pages 97-106.

[McFarling89]
Scott McFarling. *Program Optimization for Instruction Caches.* Proceedings of the 3rd International Symposium on Architectural Support for Programming Languages and Operating Systems, April 1989, pages 183-191.

[McFarlingHennessy86]
Scott McFarling and John L. Hennessy. *Reducing the Cost of Branches.* Proceedings of the 13th Annual Symposium on Computer Architecture, June 1986, pages 396-403.

[MetzgerStroud93]
Robert Metzger and Sean Stroud. *Interprocedural Constant Propagation: An Empirical Study.* ACM Letters on Programming Languages and Systems 2(1-4), March-December 1993, pages 213-232.

[Morris91]
W. G. Morris. *CCG: A Prototype Coagulating Code Generator.* Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation, June 1991, pages 45-58.

[PettisHansen90]
Karl Pettis and Robert C. Hansen. *Profile Guided Code Positioning.* Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, June 1990, pages 16-27.

[ReifLewis77]
John H. Reif and Harry R. Lewis. *Symbolic Evaluation and the Global Value Graph.* Proceedings of the 4th Annual Symposium on Principles of Programming Languages, January 1977, pages 104-118.

[Smith81]
James E. Smith. *A Study of Branch Prediction Strategies.* Proceedings of the 8th Annual Symposium on Computer Architecture, May 1981, pages 135-148.

[SteenkisteHennessy89]
Peter A. Steenkiste and John L. Hennessy. *A Simple Interprocedural Register Allocation Algorithm and Its Effectiveness for LISP.* ACM Transactions on Programming Languages and Systems 11(1), January 1989, pages 1-32.

[Wagner+94]
Tim A. Wagner, Vance Maverick, Susan L. Graham and Michael A. Harrison. *Accurate Static Estimators for Program Optimization.* Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation, June 1994, pages 85-96.

[Wall86]
David W. Wall. *Global Register Allocation at Link Time*. Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, June 1986, pages 264-275.

[Wall88]
David W. Wall. *Register Windows vs Register Allocation.* Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, June 1988, pages 67-78.

[Wall91]
David W. Wall. *Predicting Program Behavior Using Real or Estimated Profiles*. Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation, June 1991, pages 59-70.

[WegmanZadeck91]
Mark N. Wegman and F. Kenneth Zadeck. *Constant Propagation with Conditional Branches.* ACM Transactions on Programming Languages and Systems 13(2), April 1991, pages 181-210.

[WuLarus94]
Youfeng Wu and James R. Larus. *Static Branch Frequency and Program Profile Analysis.* Proceedings of the 27th International Symposium on Microarchitecture, November 1994, pages 1-11.