On the Use of Data Flow Analysis in Static Profiling*

Cathal Boogerd
Software Evolution Research Lab
Delft University of Technology
The Netherlands
c.j.boogerd@tudelft.nl

Leon Moonen
Simula Research Laboratory
Norway
Leon.Moonen@computer.org

Abstract

Static profiling is a technique that produces estimates of execution likelihoods or frequencies based on source code analysis only. It is frequently used in determining cost/benefit ratios for certain compiler optimizations. In previous work, we introduced a simple algorithm to compute execution likelihoods, based on a control flow graph and heuristic branch prediction.

In this paper we examine the benefits of using more involved analysis techniques in such a static profiler. In particular, we explore the use of value range propagation to improve the accuracy of the estimates, and we investigate the differences in estimating execution likelihoods and frequencies.

1. Introduction

Static Profiling is a technique to derive an approximated profile of a system's dynamic behavior, such as execution likelihoods or relative frequencies for various program points from source code alone. Previously, it has mainly been used in compiler optimization, to assess whether the costs associated with certain optimizations would pay off. In earlier work, we have proposed another application of such static profiles, which is to use them to prioritize the results of automatic code inspections [4].

In automatic code inspection, a tool statically analyzes code, looking for patterns or constructs that are, based on past experience, known to be fault-prone. These tools can be run early in the development phase, when it is less costly to fix defects, and thorough testing may not yet be feasible. The latter is especially relevant in our application area, software embedded in consumer electronics, as hardware may not be available early on during development.

Our research is performed in the context of the TRADER[1] project, where we investigate methods and tools to produce robust embedded software, together with our industrial partner NXP (formerly known as Philips Semiconductors). Previous experience with automatic code inspection within NXP showed that these tools typically produce too many warnings to make direct use feasible: some means of prioritization is needed. Research within TRADER focuses on defects that visibly disrupt the behavior of the system, so we want a prioritization scheme that points the developer to these defects. The simple observation that faulty code needs to be triggered, i.e., executed, in order to corrupt the system state has lead us to investigate the feasibility of using program profiling information (and thus execution likelihoods) to prioritize warning reports. The main intuition behind our approach is that the more likely it is that code having an inspection warning is executed, the higher the priority of that warning should be.

Typically, *dynamic* execution profiles are more accurate than static profiles, so they would make a logical choice. However, dynamic analysis is less suitable in our context because the embedded (soft-)real time nature of the applications makes it very hard to capture dynamic information without influencing system behavior, and system-wide dynamic analysis requires complete control over the set of test inputs, which is often lacking in the context of consumer electronics. In addition, as mentioned above, the system's hardware may not be available early on (especially for every developer). Consequently, we choose to compute a *static* execution profile for the program at hand by means of source code analysis. In essence, this means that we use the control flow context of the identified faults to assess their impact, and therefore their priority.

Another important requirement which follows from the project context is the need for a *scalable* approach: The software embedded in consumer electronics has grown tremendously over the years. For example, a modern television contains millions of lines of code, and it is these systems that our tools will need to analyze eventually. To achieve this scalability in our previous work [4], we favored elementary, conservative, techniques and applied simple heuristics to predict branching behavior. The resulting profiler uses

* This work has been carried out in the Software Evolution Research Lab at Delft University of Technology as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the BSIK03021 program.

[1] `http://www.esi.nl/trader`

only control flow information and some type information in the branch prediction heuristics.

Rather than focusing on the application, in this paper we investigate the benefits and trade-offs of using a more elaborate data flow analysis technique in such a static profiler. The technique exploits constant values present in source code in order to enhance branch prediction. We evaluate the accuracy and speed of the new static profiler with respect to our earlier approach. Specifically, the current paper investigates:

1. The amount of information that can be retrieved from constant values present in the source code.

2. The predictive strength of our profilers with regard to execution likelihood and execution frequency.

We evaluate accuracy by comparing our statically obtained results to an oracle set of values that were obtained using dynamic analysis. In contrast to what one would expect, the results show that the use of this particular data flow analysis has little impact on either the branch prediction in general or on the computed program profiles. This is surprising since earlier work by Patterson suggested that such a technique can indeed improve branch prediction [12], and hence, by extension, static profiling.

The remainder of the paper is organized as follows: Section 2 introduces methodology by summarizing our previous profiler, discussing static branch prediction techniques, and presenting the new profiler. Section 3 describes the various experiments and the experimental setup, the results of which are shown in Section 4 and summarized in Section 5. Finally, Section 6 describes related work and Section 7 concludes and discusses directions for future work.

2. Methodology

This section defines the execution likelihood and execution frequency concepts used in our investigation and discusses the techniques that we use to estimate them, such as the graph representations of source code, the graph propagation algorithms and static branch prediction schemes. We will present these concepts and techniques with respect to our previous profiler [4], and use it as a starting point for discussing the extensions that are specific to this paper.

2.1. Execution *likelihood* and *frequency*

We define *execution likelihood* of a program point v in program p as the probability that v is executed at least once in an arbitrary run of p. Similarly, we define the *execution frequency* of v in p as the average frequency of v over all possible distinct runs of p. A program point can be any expression or statement of interest, or more specifically, a vertex in the corresponding system dependence graph (SDG) [9]. We will discuss this representation later, and focus first on the difference in application of both definitions.

```
int main(int argc, char **argv) {
        /* v1 */
        int a = 1;

        foo(a);
        for (a = 0; a < argc; a++) {
                /* v2 */
        }

        foo(a);
        return 0;
}

void foo(int in) {
        /* v3 */
        if (in > 1) {
                /* v4 */
        }
}
```

Listing 1. Example of various program points that have different execution likelihoods

To understand why the distinction may be important, we will look at a few code snippets in the context of software inspection. Imagine we run a code inspection tool on the code in Listing 1, and it signals warnings in the marked locations (v_1 - v_4). We can deduce that program point v_1 must have an execution likelihood of 1, telling us that it *will always be executed in any run of the program*. A warning in such a location is obviously one that should be fixed as soon as possible, since it will certainly have an impact on the observable behavior. On the other hand, the conditionally executed v_4 will have a lower execution likelihood than v_3 because the condition may not always hold.[2] However, when looking at v_1 and v_3, it becomes difficult to distinguish between the two as they both have an execution likelihood of 1. This could become a real problem if we need to prioritize warnings in a program of which a significant part will always be executed.

Using execution *frequency* instead of execution *likelihood* solves this problem. This frequency is useful when thinking of the warnings as *potentially* leading to a corruption of the system state, and eventually a disruption of observable behavior. From this perspective, a program point with a higher frequency should be given higher priority, as with more executions, the probability of distortion increases.

Nevertheless, an intuitive difference remains between v_2, which is generally executed a number of times, and v_3, which is *guaranteed* to be executed at least once. This difference can only be properly expressed by the execution likeli-

[2] Note that we will discuss later how to compute the actual likelihood.

Information used: Prediction type:	uniform distribution assumption	condition and variable types	variable range estimates
uniform prediction	√		
heuristic based prediction	√	√	
value range propagation based prediction	√	√	√

Table 1. Branch prediction approaches and information types

hood, which is why we will assess the approaches described below for their predictive strength of *both* concepts.

2.2. Execution likelihood analysis

In previous work [4], we introduced an algorithm to compute execution likelihood based on control flow, dubbed $ELAN$ (for Execution Likelihood ANalysis). It uses a graph representation, the aforementioned SDG, a generalization of the program dependence graph (PDG). The PDG is a directed graph representing control- and data dependences within a single routine of a program (i.e. *intraprocedural*), and the SDG ties all the PDGs of a program together by modelling the *interprocedural* control- and data dependences [9]. Since $ELAN$ merely uses control flow information, we only list the most important reasons for control dependences between two vertices v_1 and v_2:

- There is a control dependence between v_1 and v_2 if v_1 is a condition, and the evaluation of v_1 determines whether v_2 is executed.
- There is a control dependence between v_1 and v_2 if v_1 is the entry vertex of function f and v_2 represents any top-level statement in f.
- There is a control dependence between v_1 and v_2 if v_1 represents a call to f and v_2 is the entry vertex of f.

Intuitively, we can find all possible acyclic execution paths by traversing the SDG with respect to control dependences. However, traversing the complete SDG to find all paths to a single point is not very efficient. To better guide this search, we base our traversals on *program slicing*.

The slice of a program p with respect to a certain location v and a variable x is the set of statements in p that may influence the value of variable x at point v. Program slices contain both control flow and data flow information. Since, at this stage, we do not need data flow information, we can restrict ourselves to control flow, and rephrase the definition as follows: the *control-slice* of v in p consists of all statements in p that determine whether execution reaches v.

Calculating the execution likelihood of v is now reduced to traversing all paths within this slice. Given the SDG of a project, we traverse the graph in reverse postorder, starting from the main function entry point (execution start) to v. For simplicity, we assume that the project contains a main function that serves as a starting point of execution, although this is not a strict prerequisite. We can pick any function f as the start point and use program chopping, an operation closely related to slicing but with given start and end points [10, 13], to compute the section of the program that can influence control, starting from f's entry point and ending at v. As the paths obtained this way are usually conditional, we need some means of *branch prediction* for all conditions on the path to compute a likelihood or frequency for a given program point v.

2.3. Branch prediction by static analysis

Various pieces of information present in the source code can provide hints as to the likelihood that a certain condition will evaluate to true or false. The different methods described below use one or more of these types of information, as has been summarized in Table 1. These methods principally use the most sophisticated information available to them to estimate branch behavior, but revert to simpler forms if the more sophisticated ones do not apply. For instance, the value range propagation method will try to use statically estimated variable values wherever possible, but uses heuristics or the uniform distribution assumption otherwise.

2.3.1. Branch prediction using uniform distribution

In the simplest form of branch prediction, we assume a uniform distribution over all branches, i.e., every branch is equally likely. This rule is applied to both conditional statements and (multi-branch) switch statements. When dealing with loops, we assume that it is more likely they are executed at least once, and use a predictive value that was determined empirically (cf. Table 2). In this approach, loops are detected by using abstract syntax tree information, while interprocedural loops (i.e. recursion) are ignored.

2.3.2. Branch prediction using heuristics

A sophistication of the simple scheme mentioned above is based on Wu and Larus' approach to branch prediction [20]. They tested a number of heuristics empirically and used the observed accuracy as a prediction for branch probability. For example, they observed that a certain heuristic predicts

Heuristic	Probability
Any loop condition	0.88
Comparison of two pointers / pointer against null	0.24
Comparison of integer to a value less than zero	0.34
Condition in loop with one branch leading out of the loop	0.31
Condition with one branch leading to a function return	0.29

Table 2. Heuristics and associated probabilities

'branch not taken' accurately 84% of the time. Therefore, when encountering a condition applicable to this heuristic, 16% and 84% are used for the 'true' and 'false' branch probabilities, respectively. In situations where more than one heuristic applies to a certain condition, the predictions are combined using the Dempster-Shafer theory of evidence [8], a generalization of bayesian theory that describes how several independent pieces of information regarding the same event can be combined into a single outcome.

One thing to note is that Wu and Larus' numbers are based on an empirical investigation of different software [1] than the system that is used in our experiments. Deitrich et al. [5] provide more insight into their effectiveness and applicability to other systems (and discuss some refinements specific to compilers). In order to fine-tune the heuristic parameters to our particular test bed, we repeated the empirical investigation, leading to slightly different values and a modest increase in accuracy. The resulting values are displayed in Table 2, for more information on these heuristics see [1].

2.3.3. Prediction using value range propagation

Estimated values for variables involved in conditions can help to model the branching behavior more accurately than using only heuristics. Previously, techniques such as constant propagation used value information present in the source code to perform, e.g., dead code elimination. Similarly, Patterson [12] uses this information in his value range propagation (VRP) technique to produce symbolic value ranges for integer variables, which in turn are used to model branching behavior. He also shows that some improvement in prediction accuracy can be achieved by only using numeric ranges. We will use a VRP approach that holds the middle ground between constant propagation and Patterson's symbolic ranges, by only using numeric ranges, in addition to other sacrifices for the sake of speed and scalability. We will outline how to compute numeric ranges as well as the differences with Patterson's original approach.

```
if (a < 1) {
        b = 1;
} else {
        b = 0;
}

if (b > 2) {
        /* dead code */
}
```

Listing 2. Example of multiple reaching definitions

VRP is an extension to the well-known constant propagation, exploiting constant (integer) values present in source code. Although constant propagation only supports a single value definition, VRP administers a range of values for every variable. This means that whereas constant propagation can only deal with one reaching definition, VRP can also handle multiple. Consider the code in Listing 2, in which a value for a cannot be statically determined: the use of b in the second condition has multiple reaching definitions, and constant propagation will therefore consider it undefined. VRP takes both definitions into account, and a subsequent analysis of the condition can tell us that the body of the second if statement is dead code. Of course, this is a contrived example, but it serves to illustrate an important difference between constant propagation and VRP.

In addition, we attach frequencies to the value ranges, which can be used to further increase the precision of branch prediction. Consider another example in Listing 3. Suppose we can establish a range for the loop variable, a, being [0:5] with frequency 6. If we further assume the actual value to be uniformly distributed across this range, we can deduce that b will have ranges of [1] with frequency 3 and [3:5] with frequency 3 at the second if statement. Therefore, we would predict that condition to be true in 50% of the cases. This

```
a = 5;
while (a >= 0) {
        if (a < 3) {
                b = 1;
        } else {
                b = a;
        }

        if (b > 2) {
                /* Some point of interest */
        }
        a--;
}
```

Listing 3. Loops and frequency-annotated ranges

also illustrates the need for support of multiple ranges with different corresponding frequencies.

This may leave the reader wondering how we compute ranges for loop counters. By inspecting the (def,kill) chains in the loop we can extract all the operations on the loop variable, and determine a single step increment or decrement. This way we can approximate the loop variable value ranges. For example, we can see that there is a single loop variable a in Listing 3, its start value is 5, the decrement is 1 (one loop operation; the post-decrement operator), and a is -1 at loop exit. The resulting range then becomes the previously mentioned [0:5] with frequency 6. Clearly, not every loop can be statically estimated this way. Loop conditions consisting of multiple variables that are changed in the loop, as well as operations that cannot be determined statically, such as random generators or memory loads, will result in an undefined range for the variables involved.

In Patterson's original VRP technique, a simple worklist algorithm is used to propagate value ranges around the control flow graph (CFG) until a fixed point is reached, both for interprocedural and intraprocedural propagation. In intraprocedural propagation, loops are handled by matching them against a template to quickly compute loop variable ranges. The algorithm reverts to the usual fixed-point computation if this matching fails.

Our approach on the other hand always uses a single-pass propagation mechanism. For intraprocedural propagation, this means handling loops in a manner similar to VRP, or failing that, using heuristics instead. For interprocedural propagation this means that recursion is ignored, gaining performance at the price of underestimation in some areas of the program. Also, as mentioned before, we use only numeric ranges instead of (limited) symbolic ranges.

Furthermore, we do not use an SSA-based graph representation, but an ordinary CFG, where the variables are made unique by distinguishing them per vertex. Moreover, our range representation holds frequencies instead of probabilities, which was done to ease the computation of execution frequencies for the corresponding vertices. These frequencies are converted on the fly to probabilities for use in branch prediction, so they do not change the results of the algorithm. Finally, both VRP approaches are context-insensitive: value ranges for formal parameters are computed by merging the ranges of the corresponding actual parameters at every call site. Patterson also employs a limited amount of procedure cloning to produce more accurate parameter ranges, however.

2.4. Execution Frequency Analysis

The graph traversal in the new profiler differs from the one in $ELAN$, as it should to be compatible with VRP. As VRP already needs to traverse the complete graph in order to estimate variable values, adding a demand-driven traversal such as in $ELAN$ makes little sense. Therefore, the frequency analysis traverses the whole graph at once, using the techniques described in the previous sections. It consists of two parts, one for interprocedural propagation, and one for intraprocedural propagation.

The interprocedural propagation starts at the 'main' function of the program, and traverses the call graph to reach other functions, administering estimates for parameters and keeping track of visited functions to prevent infinite traversals. Specifically, $InterPropagation$ consists of the following steps:

- Given inter- and intraprocedural CFGs for a program, construct a call graph, administering the number of call sites for every function in $CallPredecessors$. Similarly, record the number of in-edges for every vertex in the intraprocedural CFGs in $InEdges$. Add the 'main' function to the $InterWorkList$, and set its frequency to 1. While $InterWorkList$ is non-empty, perform the following:
    - Let F be the first item on the $InterWorkList$.
    - If all call sites in $CallPredecessors$ for F have been visited, retrieve its frequencies and the value ranges for its parameters. If the frequency is non-zero, run $IntraPropagation$, otherwise only update $CallPredecessors$ to prevent stalls in the propagation of its successors (this is in fact a performance optimization).
    - Remove F from $InterWorkList$.

Similar administration and traversal happens in the intraprocedural case; however, here we also need to evaluate expressions and conditions using the propagated estimates. In detail, $IntraPropagation$ entails:

- Let $IntraWorkList$ be a list with only the entry vertex. Save all variables and their value ranges per program point in $Bindings$, the entry vertex is assigned the frequency and parameter bindings associated with the current function. While the $IntraWorkList$ is non-empty, do:
    - Let v be the first item of $IntraWorkList$.
    - If all predecessors of v in $InEdges$ have been visited, retrieve its frequency v_f and bindings v_b and perform:
        - ∗ If all variables in the use-set of v are defined in v_b, evaluate v and update v_b for the kill-set of v.
        - ∗ If v is a condition, try to evaluate the condition using v_b, use heuristics if this fails. In both cases, update outgoing ranges to reflect the condition probabilities.

Project Name	ncKLoC	# nodes in SDG	# CPoints in SDG	Project name	ncKLoC	# nodes in SDG	# CPoints in SDG
Antiword	27	89995	2787	CCache	3	19200	541
Chktex	4	25378	769	Check	3	48825	428
Lame	27	81071	3665	Cut	1	3736	85
Link	14	67205	2969	Indent	26	35496	1554
Uni2Ascii	4	4937	141	Memwatch	2	34561	496

Table 3. Case study programs and their metrics

- ∗ If v is a call to function F, merge the bindings in v_b for all its parameters with the existing ones, and v_f to the recorded frequency for F.
- ∗ In all cases, update $Bindings$ and $Inedges$ for all successors of v in the intraprocedural CFG.

– Remove v from $IntraWorkList$.

The implementation of this algorithm is parameterized with the type of branch prediction used. This facilitates switching between heuristics and VRP and comparing the results. In the following, we will abbreviate the Execution Frequency Analysis as $EFAN$, and use a subscript H or V to indicate if respectively heuristic or VRP based branch prediction was used. Although $ELAN$ can also be instantiated with different branch predictions, we will only discuss the version using heuristics in this paper for clarity and space reasons.

3. Experimental Setup

Our investigation must show the analysis time and accuracy of the profiling techniques with respect to execution likelihood as well as execution frequency, and specifically, the applicability of VRP. Therefore, we can distinguish four seperate parts in our investigation:

IV1 An investigation into the applicability of the VRP method, showing if we can extract sufficient variable information for use in estimating branch behavior.

IV2 An evaluation of branch prediction accuracy by comparison to dynamically gathered branch behavior. This will tell us how well each technique interacts with the selected cases, and what improvement we may therefore expect in the subsequent profiler evaluation. Essentially, this is a partial repetition of the experiment in [12] on our testbed (partial, since we do not include all techniques used there).

IV3 An evaluation of profiler accuracy for both execution likelihood and frequency by comparison to dynamic profiles. In this, we want to see the impact of different branch prediction techniques (by comparing $EFAN_H$ and $EFAN_V$) and the differences in estimating likelihoods and frequencies (by comparing $ELAN$with those two). This is an extension of our previous investigation in [4].

IV4 The final evaluation is that of analysis time, which will show the cost of the additional complexity of the more elaborate techniques. Also this is an extension of our previous investigation, in techniques assessed as well as the number of cases used.

The profilers have been implemented as a plug-in for Grammatech's Codesurfer,[3] a program analysis tool that can construct control flow graphs and dependence graphs for C and C++ programs.

Case selection Table 3 lists some source code properties for the different cases, respectively the size in (non-commented) lines of code (LoC, measured using SLOCCount[4]), the size of the SDG in vertices, and the total number of conditions in the program. The apparent discrepancy between the size in LoC and the size of the SDG in vertices can be largely attributed to Codesurfer's modelling of global variables in the SDG, adding them as parameters to every function. All programs are written in C, the only language currently supported by our implementation.

The first five programs (leftmost column) were used in the accuracy evaluations (IV1-3). These programs were selected such that it would be easy to construct 'typical usage' input sets, and automatically perform a large number of test runs in order to compare our static estimates with actual dynamic behavior. For every case, at least 100 different test runs were recorded, and profile data was saved. The complete set of ten programs were used in the time evaluation (IV4) to ensure a uniform spread of datapoints.

4. Results

4.1. IV1: Applicability of VRP

Table 4 lists some metrics collected while running the VRP analysis over the accuracy testbed, i.e., the first five programs in Table 3. These numbers were aggregated over the

[3] http://www.grammatech.com/products/codesurfer/

[4] http://www.dwheeler.com/sloccount/

Metric	Number
vertices with vars	64728
vertices with int vars	27242
vertices with vars known	7482
vertices with all vars known	3162
total conditions	10331
conditions with vars known	2157
conditions with all vars known	871

Table 4. Value range propagation related metrics collected from testbed

whole accuracy testbed, and provide an indication of feasibility of using VRP. For instance, we can see from the ratio between the number of program points with integer variable occurrences and the total number of vertices with variable occurrences that limiting data flow information to integers can still yield results for a significant part of our test programs. Noteworthy is also that of those program points with integers, close to 12% can be completely statically evaluated. Finally, approximately 8% of the conditions have complete variable value information (i.e., conditions containing only integer variables, and these could all be estimated).

This last metric is the prerequisite for successful use of VRP as a branch predictor, as we can only use the value ranges for prediction in those conditions where an estimate for every variable has been determined. This does not look very promising, since there will be relatively few conditions that can benefit from the extra information VRP provides.

4.2. IV2: Branch prediction accuracy

We will assess what this means for the accuracy by comparing static estimates with their dynamic counterparts. For every branch in the test set, we produce three estimates, and check whether they are correct within a certain error margin. This is shown in Figure 1, where we can make two observations: (1) the heuristic and VRP predictors do offer improvement over the simple scheme; and (2) there is no apparent difference in accuracy between the heuristic and VRP predictors: the two lines coincide completely. Apparently the small number of conditions that could benefit from data flow information has no impact on the overall score. As this small set could include loops or branches with a great impact on the likelihood/frequency distribution, it may yet influence the accuracy of the profiler that uses the predictors.

4.3. IV3: Profiler accuracy

We evaluate profiler accuracy by looking at their *ranking* of program locations, since this is the most relevant output for typical applications. We therefore create two sets of program locations, the first sorted by one of the static profilers, the second by actual usage, and compare them using Wall's unweighted matching method [19]. This will give us a *matching score* for different sections of the two rankings. To illustrate this matching score, consider the following example: suppose we have obtained the two sorted lists of program locations, both having length N, and we want to know the score for the topmost m locations. Let k be the size of the intersection of the two lists of m topmost locations. The matching score then is k/m, where 1 denotes a perfect score, and the expected score for a random sorting will be m/N. In our experiments, scores were calculated for the topmost 10%, 20%, 30%, 40% and 50%. The rankings used are at the level of basic blocks, since there is no way for profiling to dinstinguish between locations within one block.

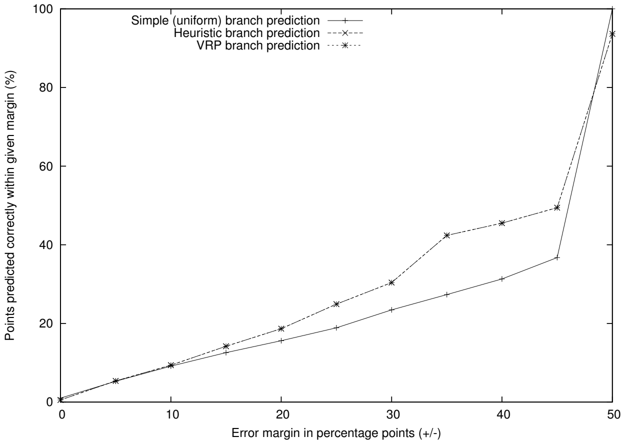

Figure 1. Branch prediction accuracy

Table 5 compares the rankings of the profilers against the dynamically obtained likelihood ranking, Table 6 compares the profilers' rankings against the dynamic frequency ranking. It may seem odd to compare a ranking based on an estimate of likelihood with a ranking based on frequency, however, both should be largely the same, modulo multiple function invocations and loop iterations. By comparing accuracy of both likelihood and frequency estimates as a predictor for frequency, we can surmise the extent to which we can statically determine the influence of these two factors.

We can make two obervations by looking at these tables: (1) using VRP does not result in more accurate estimates; and (2) $ELAN$ slightly outperforms both $EFAN$ variants, even when used as a frequency estimator. The first is in line with the earlier observed branch prediction accuracy; apparently, there is no set of 'important' branches more accurately predicted by using VRP. This is perhaps best illustrated by Figure 2, which displays the overall profiler accuracy, in a manner similar to the earlier branch prediction accuracy (Figure 1.) Once more, the lines representing the two $EFAN$ variants almost coincide. The second observation is more surprising, but may be explained by the fact that, contrary to our hope, VRP was unable to predict many

Portion	antiword			chktex			lame			link			uni2ascii		
10	0.50	0.52	0.36	0.31	0.20	0.19	0.39	0.26	0.27	0.52	0.28	0.03	0.25	0.12	0.00
20	0.46	0.42	0.38	0.54	0.39	0.38	0.38	0.25	0.25	0.58	0.40	0.24	0.71	0.53	0.29
30	0.46	0.41	0.47	0.40	0.37	0.38	0.38	0.29	0.30	0.64	0.53	0.44	0.68	0.60	0.52
40	0.47	0.46	0.52	0.39	0.34	0.41	0.46	0.42	0.45	0.65	0.64	0.58	0.65	0.50	0.47
50	0.54	0.52	0.56	0.46	0.45	0.45	0.48	0.49	0.53	0.66	0.75	0.70	0.60	0.52	0.48

Table 5. Execution Likelihood matching for $ELAN$, $EFAN_H$, and $EFAN_V$

Portion	antiword			chktex			lame			link			uni2ascii		
10	0.26	0.24	0.33	0.02	0.00	0.03	0.01	0.01	0.02	0.18	0.29	0.27	0.38	0.12	0.00
20	0.38	0.33	0.38	0.06	0.05	0.12	0.15	0.07	0.07	0.28	0.40	0.46	0.41	0.24	0.18
30	0.45	0.40	0.46	0.06	0.04	0.20	0.36	0.27	0.28	0.36	0.48	0.56	0.40	0.32	0.24
40	0.47	0.46	0.52	0.25	0.21	0.36	0.46	0.42	0.45	0.46	0.57	0.61	0.59	0.44	0.41
50	0.54	0.52	0.56	0.44	0.43	0.45	0.48	0.49	0.53	0.55	0.66	0.66	0.60	0.52	0.48

Table 6. Execution Frequency matching for $ELAN$, $EFAN_H$, and $EFAN_V$

conditions, most particularly loops. In such cases a simple fixed estimate for loop iterations is not a good alternative. Although not shown here, we ran multiple experiments to find a relation between several structural properties of the loops and the corresponding number of iterations, but none were found. In likelihood estimates loops are less dominant, and therefore their impact on the accuracy of the estimates.

4.4. IV4: Profiler analysis time

With regard to the speed of the profilers, we are primarily interested in the difference between $EFAN_H$ and $EFAN_V$, as we investigated the time performance behavior of $ELAN$ before. We have seen that the kind of operations and the single-pass traversal involved are similar to $ELAN$. Thus, we also expect their time characteristics to be similar to that profiler, which was found to be approximately linear in the size of the SDG. Note that there is a difference in the traversal; while $ELAN$ traverses over dependence edges, $EFAN$ uses control flow edges, probably visiting more vertices as a result. Figure 3 displays analysis time measurements for the different programs, and in fact we can observe that $EFAN_H$ and $EFAN_V$ both behave similar to $ELAN$ (cf. Figure 2 in [4]). Two programs do not fit the trend line, they were found to have some functions of a highly branching nature (e.g., having switches of 500-1000 cases).

5. Evaluation

Returning to our four lines of investigation, we can summarize the results as follows:

IV1 The numeric VRP approach can estimate value ranges for approximately 12% of all locations containing integer variables, and 8% of all conditions containing integer variables. This constitutes an upper bound on the number of conditions that may be estimated using numeric VRP, as there are other structural requirements

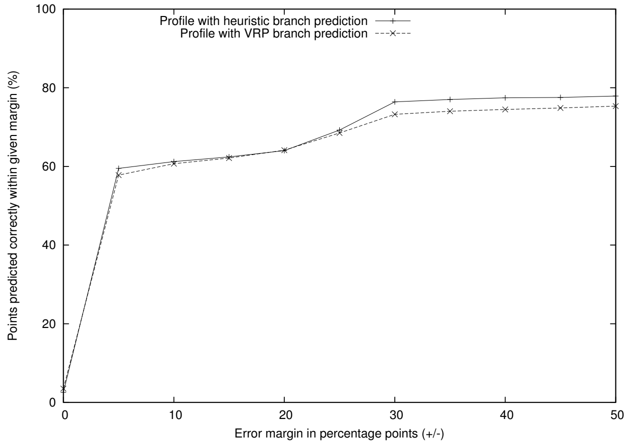

Figure 2. Profile prediction accuracy

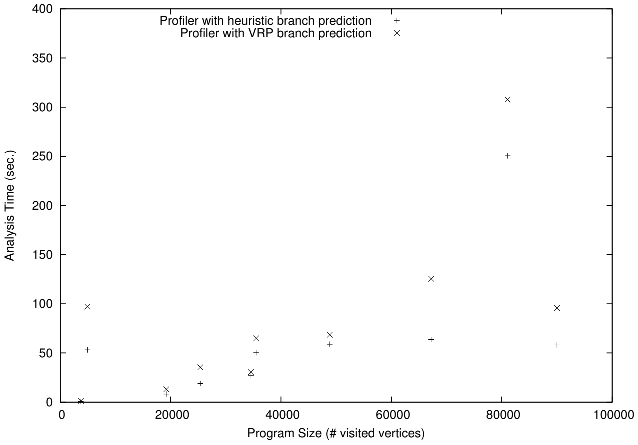

Figure 3. Profiler analysis time

that need to be met. For instance, also including non-integer values or function calls in the condition will render the approach ineffective.

IV2 As a result of said limited applicability, VRP performs on par with the heuristic branch prediction approach in terms of accuracy. This result contrasts with Patterson's findings on numeric ranges, which did show an improvement over heuristics [12]. Two main differences between his approach and ours that may be responsible are fixed-point calculation instead of single-pass traversal, and the use of procedure cloning. The first is likely to benefit the *accuracy* of branches in functions part of a recursion, but will not increase the *number* of branches where VRP is applicable. Procedure cloning helps where parameters of a function have a known definition in one calling context, but not in another. The current context-insensitive approach will thus be unable to approximate the parameter value, but will be able to distinguish the two reaching definitions in case of cloning. Therefore, this seems the most likely culprit for the difference in findings.

IV3 Although a more accurate estimate of a limited number of high-impact branches could positively influence the accuracy of the static profiler, this does not appear to be the case either. Perhaps more surprising is the fact that $ELAN$ outperforms both $EFAN$ variants not only in estimating execution likelihood, but als for execution frequency. The most likely reason for this is the loop estimation in the $EFAN$ approaches, which both need to resort to a generic value, whereas the value used in $ELAN$ has previously shown itself well-suited to our testbed [4]. In addition, in likelihood estimation the values used for loops have a less prominent effect on the overall likelihood estimate than the values for loop iteration have on overall frequency estimates.

IV4 The profiling techniques and the branch predictions employed were all designed for simplicity and scalability, and results of the analysis time investigation reflect this. Notably, the numeric VRP method only imposes a linear-time overhead on the profilers with less elaborate techniques (i.e., heuristics and uniform prediction).

Internal validity As the approaches in this paper are based on the CFG and SDG, the way in which these graphs are constructed directly affects the outcome, especially in terms of accuracy. It should be noted, therefore, that the graphs both have missing dependences (false negatives) and dependences that are actually impossible (false positives). For example, control- or data dependences that occur when using setjmp/longjmp are not modeled. Another important issue is the accuracy of dependences in the face of pointers, think for example of modeling control dependences when using function pointers. To improve this accuracy, a flow insensitive and context insensitive points-to algorithm [15] is employed to derive safe information for every pointer in the program. This means that our profilers also model control flow through pointers (i.e., function pointers) and data flow through pointers to integer variables.

External validity Our testbed consists of programs adhering to a simple input-output paradigm, which makes creation of appropriate test inputs easier, and allows us to focus on evaluating the approach itself. Generalizing to other kinds of programs may not be too hard, as the approaches are based on control- and data-flow information, which will always be present in any program. Still, there may be an increased influence of interaction or inputs in comparison to programs in our testbed. However, Fisher and Freudenberger observed that, in general, varying program input tends to influence which parts of the system will be executed, rather than influencing the behavior of individual branches [7]. This suggests that branch behavior is no different in other types of programs, and that static profiling should be similarly applicable elsewhere.

6. Related Work

Static profiling is used in a number of compiler optimizations and worst-case execution time (WCET) analyses, aiming at identifying heavily-executed portions of the code. Branch prediction in these techniques often takes the form of heuristics [20, 18]. A more sophisticated data flow based approach is described by Patterson who uses statically derived value ranges for variables to predict branching behavior [12]. Closely related techniques range from constant propagation [11, 14, 2], to symbolic range propagation [16, 3], or even symbolic evaluation [6]. However, none of these techniques has previously been applied to static branch prediction or in the computation of static profiles.

Voas [17] proposed the concepts of execution probability, infection probability and propagation probability to model the likelihood that a defect leads to a failure. The first of these is similar to our notion of execution likelihood: the chance that a certain location is executed. However, the application is quite different, in the sense that Voas explicitly looks for those areas where bugs could be hiding during testing, whereas we are looking for 'visible' bugs out of a list too large to solve. Moreover, Voas computes these metrics by means of a *dynamic* analysis, which is less suitable for our application area, as discussed earlier.

7. Concluding Remarks

Contributions In this paper, we have (1) presented a novel approach to use data flow information for computing static profiles, by using statically derived value ranges for integer variables in branch prediction; (2) identified the merits of estimating both execution likelihood and frequency; and (3) assessed the new approach with respect to both concepts in comparison to our earlier static profiler.

Conclusions Although we found that some branches in our testbed potentially benefit from VRP information, the actual number of branches that could be predicted this way is limited. In addition, we found that the simple control-dependence based solution, $ELAN$, outperformed both $EFAN$ variants in estimating both likelihood and frequency, most likely because the value range information is insufficient to accurately model loop behavior. We conclude that the use of these more involved analyses has little impact on the accuracy of either branch prediction or profiling.

Future Work Patterson's work suggests that branch prediction can be further improved using simple *symbolic* ranges instead of purely numeric ones [12]. An interesting direction for future work is therefore to extend the current profilers with some form of symbolic evaluation and investigate the benefits for accuracy and the costs in terms of additional performance overhead.

References

[1] T. Ball and J. R. Larus. Branch Prediction For Free. In *ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 300–313, June 1993.

[2] D. Binkley. Interprocedural Constant Propagation using Dependence Graphs and a Data-Flow Model. In *5th Int. Conf. on Compiler Construction (CC)*, volume 786 of *LNCS*, pages 374–388. Springer, 1994.

[3] W. Blume and R. Eigenmann. Demand-Driven, Symbolic Range Propagation. In *8th Int. Ws. on Languages and Compilers for Parallel Computing, (LCPC'95)*, volume 1033 of *LNCS*, pages 141–160. Springer, 1995.

[4] C. Boogerd and L. Moonen. Prioritizing Software Inspection Results using Static Profiling. In *Proc. Sixth IEEE Int. Ws. on Source Code Analysis and Manipulation (SCAM)*, pages 149–158. IEEE CS, September 2006.

[5] B.L. Deitrich, B-C. Cheng, and W.W. Hwu. Improving static branch prediction in a compiler. In *18th Ann. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 214–221. IEEE CS, 1998.

[6] T. Fahringer and B. Scholz. A Unified Symbolic Evaluation Framework for Parallelizing Compilers. *IEEE Trans. Parallel Distrib. Syst.*, 11(11):1105–1125, 2000.

[7] J.A. Fisher and S.M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, 1992.

[8] Shafer G. *A Mathematical Theory of Evidence*. Princeton Univ. Press, 1976.

[9] S. Horwitz, T.W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[10] D. Jackson and E. J. Rollins. A new model of program dependences for reverse engineering. In *Proc. 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering*, pages 2–10. ACM Press, 1994.

[11] J. Knoop and O. Rüthing. Constant propagation on the value graph: Simple constants and beyond. In *9th Int. Conf. on Compiler Construction (CC)*, volume 1781 of *LNCS*, pages 94–109. Springer, 2000.

[12] J. R. C. Patterson. Accurate static branch prediction by value range propagation. In *ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 67–78, 1995.

[13] T. W. Reps and G. Rosay. Precise interprocedural chopping. In *Proc. 3nd ACM SIGSOFT Symp. on Foundations of Software Engineering*, pages 41–52. ACM Press, 1995.

[14] S. Sagiv, T. W. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *6th Int. Conf. on Theory and Practice of Software Development (TAPSOFT'95)*, volume 915 of *LNCS*, pages 651–665. Springer, 1995.

[15] M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 1–14, January 1997.

[16] C. Verbrugge, P. Co, and L.J. Hendren. Generalized constant propagation: A study in C. In *6th Int. Conf. on Compiler Construction (CC)*, volume 1060 of *LNCS*, pages 74–90. Springer, 1996.

[17] J.M. Voas and K.W. Miller. Software Testability: The New Verification. *IEEE Softw.*, pages 17–28, 1995.

[18] T.A. Wagner, V. Maverick, S.L. Graham, and M.A. Harrison. Accurate static estimators for program optimization. In *ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 85–96, 1994.

[19] D.W. Wall. Predicting program behavior using real or estimated profiles. In *ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 59–70, June 1991.

[20] Y. Wu and J.R. Larus. Static branch frequency and program profile analysis. In *27th Ann. Int. Symp. on Microarchitecture*, pages 1–11. ACM/IEEE, 1994.