Goal-Directed Value Profiling

Scott Watterson Saumya Debray

Department of Computer Science
University of Arizona
Tucson, AZ 85721
{saw, debray}@cs.arizona.edu

Abstract. Compilers can exploit knowledge that a variable has a fixed known value at a program point for optimizations such as code specialization and constant folding. Recent work has shown that it is possible to take advantage of such optimizations, and thereby obtain significant performance improvements, even if a variable cannot be statically guaranteed to have a fixed constant value. To do this profitably, however, it is necessary to take into account information about the runtime distribution of values taken on by variables. This information can be obtained though value profiling. Unfortunately, existing approaches to value profiling incur high overheads, primarily because profiling is carried out without consideration for the way in which the resulting information will be used. In this paper, we describe an approach to reduce the cost of value profiling by making the value profiler aware of the utility of the value profiles being gathered. This allows our profiler to avoid wasting resources where the profile can be guaranteed to not be useful for optimization. This results in significant reductions in both the time and space requirements for value profiling. Our approach, implemented in the context of the *alto* link-time optimizer, is an order of magnitude faster, and uses about 5% of the space, of a straightforward implementation.

1 Introduction

Compilers can exploit knowledge that an expression in a program can be guaranteed to evaluate to some particular constant at compile time via the optimization known as constant folding [14]. This is an "all-or-nothing" transformation, however, in the sense that unless the compiler is able to guarantee that the expression under consideration evaluates to a compile-time constant, the transformation cannot be applied. In practice, it is often the case that an expression at a point in a program takes on a particular value "most of the time" [5]. As an example, in the SPEC-95 benchmark *perl*, the function *memmove* is called close to 24 million times: in almost every case, the argument giving the size of the memory region to be processed has the value 1; we can take advantage of this fact to direct such calls to an optimized version of the function that is significantly simpler and faster. As another example, in the SPEC-95 benchmark *li*, a very frequently called function, *livecar*, contains a `switch` statement where one of the case labels, corresponding to the type `LIST`, occurs over 80% of the time; knowledge of this fact allows the code to be restructured so that this common case can be tested separately first, and so does not have to go through the jump table, which is relatively expensive. As these examples suggest, if we know that certain values occur very frequently at certain program points, we may be able to take advantage of this information to improve the performance of the program. This information is given by a *value profile*, which is a (partial) probability distribution on the values taken on by a variable when control reaches the program point under consideration at runtime. Our experience with value-profile-based optimizations indicate that they can produce significant speedups for non-trivial programs [15].

Unfortunately, existing approaches to obtaining value profiles tend to be very expensive, both in time and space. For example, in an implementation of "straightforward" value profiling by Calder *et al.*, executables instrumented for value profiling are more than 30 times slower, on the average, than the uninstrumented executables. The reason for this is that their profiling decisions are made without any knowledge of the potential utility of the profiles, i.e., the ways in which the profiles will be used for optimization. Our experiments indicate that of all the variables and program points that are candidates for value profiling in a program, typically only a tiny fraction actually yield value profiles that lead to profitable optimizations.[1] This means that in value profilers based on existing techniques, much of the overhead incurred in the course of profiling represents wasted work.

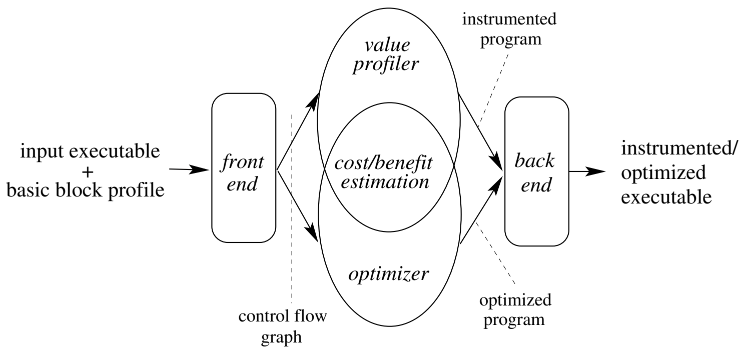

Fig. 1. *Alto* system structure

In this paper, we describe a system that avoids these problems by tightly integrating the value profiler with the optimizer that uses the results of value profiling (see Figure 1). This gives the value profiler a mechanism for estimating the utility of a value profile at a given program point, and allows it to identify profiling candidates—i.e., variables (memory locations, registers) at particular program points—that are clearly not worth profiling, i.e., that can be guaranteed to not yield optimization opportunities. This is done both during the instrumentation phase, when the profiler decides where to insert instrumentation code for value profiling, and during the actual profiling itself, when the instrumentation code so inserted can determine whether it is worth continuing to pro-

[1] A significant reason for this is that, as discussed in Section 4, code optimized to exploit runtime value distributions must be guarded by a test, and the cost of this runtime test must be taken into account when weighing the profitability of the optimization.

file a particular profiling candidate. By being able to use the cost-benefit considerations guiding the optimizer, the profiler is able to identify and prune value profiling candidates that have no possibility of contributing to any optimization. The resulting system is highly selective: typically, fewer than 1% of all possible profiling candidates are chosen for actual profiling. This selectivity leads to significant improvements in profiling performance, yielding order-of-magnitude improvements in both the space and time requirements for profiling, while retaining all of the optimization opportunities.

There has been a great deal of work on profiling techniques [2, 3, 13] and profile-guided code optimization [6, 7, 17]. However, to the best of our knowledge this is the first work that aims to make the profiler *goal-directed*, i.e., aware of the potential utility of the profiling information that it is gathering, and able to modify its profiling actions based on this awareness, in order to reduce profiling overheads.

2 System Overview

The work described here has been implemented in the context of *alto*, a link-time optimizer we have constructed for the Compaq Alpha architecture [16]. *Alto* can be used to rewrite executable files, either to instrument them for profiling, or to carry out code optimization. If profiling is carried out, a command line option can be used to control whether basic block profiles, edge profiles, or value profiles are gathered.

This paper focuses on value profiling. The gathering of value profiles is a two-stage process. We first use *alto* to instrument the input program—say, *a.out*—for basic block profiling. This yields an instrumented executable *a.bblprof.out* that is then executed using representative "training inputs" to obtain a basic block profile. Using this basic block profile, the original executable *a.out* is instrumented again, this time to gather value profiles. This yields a second instrumented executable *a.valprof.out*. Finally, *a.valprof.out* is executed with the training input to obtain a value profile.

For subsequent optimization, the basic block and value profiles so gathered are fed back to *alto*, which uses them to apply various code transformations on the input program *a.out*.

3 Mechanics of Value Profiling

Our value profiling technique is inspired by (and largely based upon) work by Calder *et al.*[5]. We keep track of the top N values occurring in a given register r at a given program point p. In addition to the top N values, we include a field that counts the number of values that are not recorded in the table; we label this the *other* field. Table 1 shows an example of a value profile, where N = 6. The field labelled *other* refers to values other than 1, 9, 20, 23, 100, or 0.

Each time execution reaches p, we profile the value contained in register r. If the value is in the table, we increment the count of that value. If the value contained in r is not already in the table, then we attempt to insert it. If there is no room in the table for the new value, we increment the count of the *other* field. In the above example table, if register r contains the value 100, the count of 100 in the table would simply be incremented. If r contained the value 2, then the count of *other* would be incremented, since there is no room in the table to insert 2.

Value	*Count*
1	2635
9	1093
20	1395
23	244
100	140
0	410
other	4698

Table 1. Example Value Profile Table

The number of entries in the value profiling table has a significant impact on the running time of the value profiler. The table must be large enough that the most frequent entry will be in the table, but as the table grows, the profiler must look at more entries each time it reaches a profiling point. In this paper, we chose a value profiling table with 6 entries, plus the *other* field. This table size was the smallest size that captured most optimization opportunities. Further analysis of table sizes can be found in [10].

In some cases, the most frequently occurring value overall is not one of the first N distinct values. We would still like to capture this value during value profiling. We provide a mechanism for allowing later values into the table by periodically cleaning the lower half of the table. This means that the values are sorted based on their counts, and the N/2 least frequently occurring values are evicted (their counts are added to the *other* count). Though cleaning is a fairly heavy process, it happens infrequently. Our experiments indicate that, except for very small values of the cleaning interval, the actual cleaning frequency does not significantly affect either speed or quality of profiling. Given this insensitivity, we chose—in part to simplify comparison of results—the same cleaning interval as Calder *et al.* [5], namely, 1000. In other words, the table is cleaned after the execution has reached p 1000 times. After a table has been cleaned once, we must make sure that a new value can enter the steady part of the table before we clean again. We set the new value of the cleaning interval to 1000 + the count of the last entry in the steady part. This means that if a new value occurs almost exclusively between cleanings, its count should be higher than the last entry in the steady part of the table. It will then be moved into the steady part, evicting the least frequently encountered value.

Each program point selected for profiling incurs both an execution and a space cost. Profiling every register at every program point is obviously wasteful, so we must be more selective. Calder *et. al* instrumented every load instruction. We consider this the baseline approach, and discuss it in more detail in Section 5.1. Section 4 describes our approach.

4 Goal-Directed Value Profiling

To minimize the amount of wasted work during value profiling, we attempt to detect, as early as possible, those profiling candidates whose value profiles can be guaranteed to not yield useful optimizations. There are two possibilities for the detection of such unprofitable profiling candidates:

(i) We may be able to tell, based on the basic block execution profile of the program, that specializing the program based on the values of a variable at a given program point will not yield an improvement in performance, regardless of the value profile for the variable at that program point. In this case, we can avoid profiling this candidate altogether. Section 4.1 describes a cost model for value-profile-based code specialization, and Section 4.3 discusses how this model can be used for a cost-benefit analysis that allows us to avoid profiling unprofitable candidates.

(ii) Even if we cannot eliminate a profiling candidate ahead of time as discussed above, we may find, as profiling progresses, that the value profile for a particular profiling candidate is simply not "good enough" to yield any profitable optimization, regardless of the values that may be encountered for that candidate during the remainder of the computation. When this happens, we can discontinue profiling this candidate. A natural place to do this is at the point where a value profile table is cleaned, since cleaning requires us to examine the table in any case. The details of how we determine whether it is worth continuing to profile a particular candidate are discussed in Section 4.4.

Calder *et. al* note that instructions other than loads may be highly invariant [4]. For example, consider the following code:

r1 $\leftarrow$ load 0(r18)
r2 $\leftarrow$ and r1, 0xf

In this sequence, the `and` instruction may always produce the same result (if all of the loads have the same low 4 bits) despite the load instruction never loading the same value twice.

This discovery makes value profiling potentially much more powerful. The optimizer can consider many potentially useful program points other than load instructions for specialization. These may include function arguments (profiled at function entry), switch statement selectors, and other such values. Calder reports that full profiling for all load instructions results in a 32x slowdown. Since there are many program points, our cost-benefit analysis is of critical importance in reducing the amount of time and space used for profiling. We consider all registers at all program points candidates, and use the cost-benefit analysis as discussed in Sec 4.3 to choose those candidates that may yield useful optimizations.

Our system focuses on code specialization, since that optimization is implemented in `alto`. If we change the optimizations used in `alto`, for instance to reduce indirect function call overhead [18, 19], we need not change the profiler. Since the optimizer shares the cost-benefit decisions with the profiler, the profiler would simply select different program points, based on the expected benefit.

4.1 A Cost Model for Value-Profile-Based Code Specialization

Our approach uses value profiles primarily for value-based specialization. By this we mean the elimination of computations whose result can be evaluated statically. Suppose we have a code fragment C that we wish to specialize for a particular value v of a register (or variable) `r`. Conceptually, value-profile-based specialization transforms the code to have the structure **if** (r == v) **then** $\langle C\rangle_{\mathtt{r}=v}$ **else** C where $\langle C\rangle_{\mathtt{r}=v}$ represents the residual code of C after it has been specialized to the value v of `r`. The test '**if** `(r == v) ...`' is needed because the value of r is not a compile-time constant, i.e., we cannot guarantee that `r` will not take on any value other than v at that program point.

Notice that, while the specialized code $\langle C\rangle_{\mathtt{r}=v}$ may be more efficient than the original code C, the overall transformed code will actually be less efficient than the original code for values of `r` other than v because of the runtime test that has been introduced. There is thus a tradeoff associated with the transformation: if the optimized code is not executed sufficiently frequently, then the cost of performing the runtime test will outweigh the benefit of performing the value-based optimizations.

This optimization therefore requires a cost-benefit analysis to determine when to specialize a program. Our analysis attempts to assign a benefit value to each instruction. The benefit computation assigns a value to a ⟨program point, register⟩ pair. The benefit is an estimate of the savings from specializing code based on knowing the value of the register r at the program point p. This benefit computation has two components:

(i) For each instruction I that uses the value of r available at p, there may be some benefit to knowing this value. The magnitude of this benefit will depend on the type of I, and is denoted by $\mathsf{Savings}(I,r)$.

(ii) It may happen that knowing the value of an operand register of an instruction allows us to determine the value computed by I. In this case, I is said to be *evaluable* given r. If I is evaluable given r, the benefit obtained from specializing other instructions that use the value computed by I are also credited to knowing the value of r at p. The indirect benefits so obtained from knowing the value of r in instruction I are denoted by $\mathsf{IndirBenefit}(I,r)$.

The savings obtained from knowing the operand values for an individual instruction is essentially the latency of that instruction, if knowing the operand values allows us to determine the value computed by that instruction, and thereby eliminate that instruction entirely[2] (our implementation uses latency figures for various classes of operations based on data from the Alpha 21164 hardware reference manual):

$$\mathsf{Savings}(I,r) = \textbf{if } \mathsf{Evaluable}(I,r) \textbf{ then } \mathsf{Latency}(I) \textbf{ else } 0.$$

Let $\mathsf{Uses}(p,r)$ denote the set of all instructions that use the value of register r that is available at program point p. Then the benefit of knowing the value of a register r at program point p is given by the following:

$$\mathsf{Benefit}(p,r) \quad = \sum_{I\in\mathsf{Uses}(p,r)} (\mathsf{ExecutionFreq}(I) \times \mathsf{Savings}(I,r) + \mathsf{IndirBenefit}(I,r))$$

$$\mathsf{IndirBenefit}(I,r) = \textbf{if } \mathsf{Evaluable}(I,r) \textbf{ then } \mathsf{Benefit}(p', \mathit{ResultReg}(I)) \textbf{ else } 0.$$

[2] The benefit estimation can be improved to take into account the fact that for some instructions, knowing some of the operands of the instruction may allow us to strength-reduce the instruction to something cheaper even if its computation cannot be eliminated entirely. While our implementation uses such information in its benefit estimation, we don't pursue the details here due to space constraints.

Here p' is the program point immediately after I, and $ResultReg(I)$ the register into which I computes its result.

The equations for computing benefits propagate information from the uses of a register to its definitions. These equations may be recursive in general, since there may be cycles in the use-definition chain. `alto` computes an approximation of the solution to the benefit equations. This estimate is used in `alto` to select program points for value-based optimizations.

We use a simple decision function for estimating when the net benefit due to specialization for a given value profile is high enough to justify specialization for a particular value v:

$$\mathsf{Benefit}(p,r) \times \mathsf{prob}(v) - \mathsf{TestCost}(r,v) \times \mathsf{ExecutionFreq}(p) \geq \phi, \tag{1}$$

where $\mathsf{prob}(v)$ is the probability of occurrence of the value v, $\mathsf{TestCost}(r,v)$ denotes the cost of testing whether a register r has a value v, [3] and ϕ is an empirically chosen threshold. This means that we will choose program point p and and register r for optimization if the benefit is high enough. Note that the cost of testing for the value v is taken into account by subtracting it from the benefit.

This benefit computation is somewhat limited. It does not consider different possible values when computing the benefit for knowing the result of an instruction. In some cases (such as if the value is always zero), this means our computation may miss some opportunities to greatly simplify the code. Using particular values, however, results in very high overhead for the cost-benefit computation. Our computation also does not attempt to precisely model the cache behavior of the specialized code. We considered several alternative computations, but most were either impractical or too costly to compute. Our benefit computation, while imprecise in some respects can be computed quickly, and produces noticeable speedup on significant benchmarks (see Figure 6).

4.2 Expression Profiling

The idea of value profiling can be generalized to that of *expression profiling*, where we profile the distribution of values for an arbitrary expression, not just a variable or register, at a given program point. Examples include arithmetic expressions, such as "the difference between the contents of registers r_a and r_b" and boolean expressions such as "the value of register r_a is different from that of register r_b" In general, the expressions profiled may not even occur in the program, either at the source or executable level.

Expression profiles are not simply summaries of value profiles: e.g., given value profiles for registers r_a and r_b, we cannot in general reconstruct how often the boolean expression $r_a == r_b$ holds. Expression profiles are important for two reasons. First, they conceptually generalize the notion of value profiles by allowing us to capture the distribution of relationships between different program entities. Second, an expression profile may have a skewed distribution, and therefore enable optimizations, even if the value profiles for the constituents of the expression profile are not very skewed: for

[3] The cost of the test varies, depending on the value of v. Usually, this test requires at least two instructions, a compare and a branch. If no registers are free, the test cost is higher. The cheapest value to test against is zero, since that requires only a conditional instruction.

example, a boolean expression $r_a \neq r_b$ may be true almost all of the time even if the values in r_a and r_b do not have a very skewed distribution.

The expressions that we choose to profile are determined by considerations of the optimizations that they might enable. Our implementation currently targets two optimizations: *loop unrolling* and *load avoidance*. More detailed discussion of expression profiling can be found in [15].

4.3 Static Elimination of Unprofitable Candidates

When determining whether to profile a particular profiling candidate, we consider whether the net benefit from any specialization for that candidate would be high enough in the best possible case. For this we assume that $\mathsf{prob}(v) = 1$ (i.e., v is the only value encountered at runtime), and $\mathsf{TestCost}(r, v)$ is the cheapest possible, i.e., we test against the value 0. If, despite these optimistic assumptions, the benefit of performing optimization is outweighed by the cost of performing the test, this variable is eliminated as a candidate for profiling. In other words, a program point register pair $\langle p, r \rangle$ is selected for profiling if and only if,

$$\mathsf{Benefit}(p, r) \geq \mathsf{TestCost}(r, 0) \times \mathsf{ExecutionFreq}(p) + \phi, \tag{2}$$

Using this selection algorithm has the additional property that profiling will tend to be inserted in the less frequently executed portions of the code. Inserting tests in the middle of a frequently executed loop is unlikely to result in speedups, and using a cost benefit model accurately predicts that profiling such an instruction would be a waste of resources.

This elimination of candidates is conservative, in the sense that no matter what distribution of values is observed at runtime, the compiler will never choose this candidate for specialization. This means that profiling these instructions would be a waste of resources. This elimination of candidates is also very effective: As shown in Table 2, fewer than 1% of possible program points are selected for profiling. This results in a reduction in execution time of 74% on average (see Figure 2).

Program	*No. of Program Points*			
	Total	*Baseline*	*Static*	*Dynamic*
compress	16749	3302	71	17
gcc	271899	32910	6459	871
go	65328	14710	1273	262
ijpeg	49650	11728	189	31
li	32221	5404	152	33
m88ksim	40867	7535	211	54
perl	82462	16500	430	160
vortex	113236	23499	242	36

Table 2. Program Points Profiled Using Different Techniques

4.4 Dynamic Elimination of Unprofitable Candidates

Given the cost-benefit computation `alto` uses for selecting optimization opportunities, we can determine before value profiling how frequently the most frequent value must appear in order to perform specialization. This frequency can be used to turn off profiling when it becomes unprofitable (i.e. the compiler will not perform the optimization). The cutoff threshold is computed by taking equation 1 and solving for $\mathsf{prob}(v)$.

$$\mathsf{prob}(v) \geq \phi + \frac{\mathsf{TestCost}(r,0) \times \mathsf{ExecutionFreq}(p)}{\mathsf{Benefit}(p,r)}, \tag{3}$$

This gives a lower bound on the probability the most frequent value must have in order for the optimizer to perform specialization. Since we know the execution frequency of p from our basic block profile, we can substitute that into the equation for $\mathsf{prob}(v)$, yielding the following:

$$\frac{\mathsf{count}(v)}{\mathsf{ExecutionFreq}(p)} \geq \phi + \frac{\mathsf{TestCost}(r,0) \times \mathsf{ExecutionFreq}(p)}{\mathsf{Benefit}(p,r)},$$

If we solve this formula for $\mathsf{count}(v)$, we can compute the minimum count that the most frequently occurring value must have for the optimizer to perform specialization. We then compute our threshold by subtracting this minimum count from the execution frequency of p. We know that if the count of the *other* field ever exceeds this threshold, then the optimizer can not select this value for specialization (since $\mathsf{count}(v)$ can not be high enough)[4]. This means, that if the *other* threshold is exceeded, we should stop profiling this point, since it will no longer be selected for optimization.

The cutoff threshold is thus computed as follows (ϕ is the same empirically chosen threshold as in equation 1):

$$Threshold = (1 - \frac{\phi + \mathsf{TestCost}(r,0) \times \mathsf{ExecutionFreq}(p)}{\mathsf{Benefit}(p,r)}) \times \mathsf{ExecutionFreq}(p).$$

Each profiling table has a boolean flag added, which determines when value profiling takes place. If the flag is true, normal value profiling takes place, if not, the value profiling is skipped, and execution is returned to the original code. This boolean flag is checked at every execution of the original profiling point.

The boolean flag is set to true before profiling begins. Each time the value profiling table is cleaned, the table is inspected to see if profiling should continue. The count of the *other* field is compared to the threshold cutoff value, and if the threshold is exceeded, the boolean flag is set to false. Thereafter, value profiling will not be performed at this program point.

Use of the cutoff threshold as computed above results in an additional 25% reduction in execution time beyond static elimination (see Figure 2). Again, this threshold is a conservative choice, since the compiler would never choose this instruction for optimization after exceeding the threshold as computed above.

---

[4] Strictly speaking, profiling could stop when the sum of all table entries other than the most frequent value exceed the threshold. For reasons of efficiency and simplicity of implementation, we consider the more conservative criterion given above.

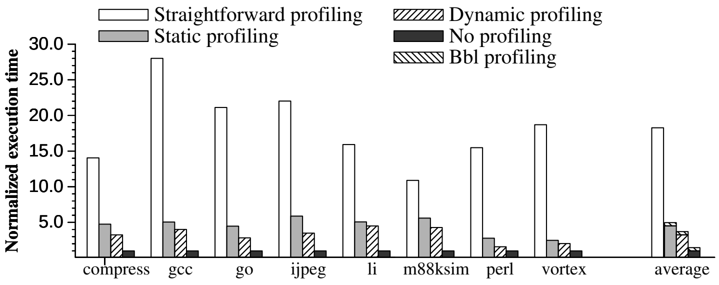

Fig. 2. Impact of Value Profiling on Execution Time (Relative to No Profiling)

5 Performance

We implemented our value profiling approach within the `alto` link-time optimizer. The programs used were the eight SPEC-95 integer benchmarks. The programs were compiled with the vendor supplied C compiler at maximum optimization (cc -O4) with additional flags to produce statically linked executables.

Our tests used the training input for all benchmarks. Times reported represent the average of ten runs, with the fastest and slowest times eliminated.

5.1 Baseline

Calder *et al.* instrument the register result of every load instruction [5]. To obtain a baseline for our results, we implemented this approach in `alto`. Table 2 shows the number of load instructions profiled using this approach. We implemented this straightforward approach only for comparison purposes.

In many cases such a straightforward approach will waste significant time and space. Most of the instructions being profiled will never be chosen by the optimizer for value-based optimizations. Profiling these points is a waste of resources, since the optimizer does not make use of the information obtained by the profiler. Knowing how the optimizer will make use of the profiles allows us to focus our profiler on those program points that may be chosen for value-based optimization.

5.2 Experimental Results

Figure 2 shows the performance results, where No profiling represents the running time of the original executable. The Straightforward profiler profiled the result of every load instruction, as discussed in section 5.1. The Static profiler used static elimination of unprofitable candidates, as discussed in section 4.3. The Dynamic profiler added dynamic elimination of candidates as discussed in section 4.4.

Our baseline implementation profiled the result of every load instruction. This was quite costly, both in time and space, with a slowdown of 10-28 times the original code. After eliminating candidates unsuitable for profiling, the cost of profiling was reduced to 2.4-5x of the original execution. Space overhead was also significantly reduced,

Program	*Profiling Space* (Bytes)		$Size_{Straightforward}/Size_{opt}$
	Straightforward	Optimized	
compress	343408	7384	0.0215
gcc	3422640	671736	0.1960
go	1529840	132392	0.0865
ijpeg	1219712	19656	0.0161
li	583632	15808	0.0270
m88ksim	783640	21944	0.0280
perl	1716000	44720	0.0260
vortex	2443896	25168	0.0102
average	1231447	134115	0.0516

Table 3. Space Requirements for Profiling

on average our approach uses 5% of the space of the straightforward approach. After adding the threshold optimization, the cost of value profiling was reduced to 1.5-4.5x of the original execution time. Table 3 shows the space overhead of value profiling.

The last column for Figure 2 shows the geometric mean of performance for each implementation. Above each bar we also show the runtime overhead of gathering basic block profiles, since we use this information to select value profiling opportunities. It can be seen that this additional overhead of basic block profiling is very small. This information would be gathered for optimization purposes no matter which implementation of value profiling is used.

5.3 Low Level Characteristics

We examined the performance of our profiling code using hardware performance counters. Unsurprisingly, we found that the Straightforward profiler used many more cycles than either the Static or Dynamic profilers. This is due to instrumenting many more program points than the other two approaches, which results in a much higher instruction count for the Straightforward profiler. Figure 3 shows the cycles for the original code, as well as the three profilers.

Adding the profiling cutoff threshold to the Static profiler resulted in a significant drop cycles and instructions. This is due to the Dynamic profiler abandoning program points when they would no longer be selected by the optimizer. Figure 4 shows the number of loads executed, and Figure 5 shows the number of branch instructions executed.

For some benchmarks, such as compress, the profiling cutoff threshold was extremely effective. Many of the program points profiled were either extremely unpredictable (resulting in the cutoff being invoked quickly) or extremely predictable (resulting in the most frequent value being the initial table entry). For other benchmarks, the cutoff was less useful. This is largely due to the cleaning check only considering the *other* field when turning on or off profiling. If the value table has 3 values that each happen 33% of the time, then the cutoff will not be invoked. However, the flag is still checked every time the value profiling is invoked.

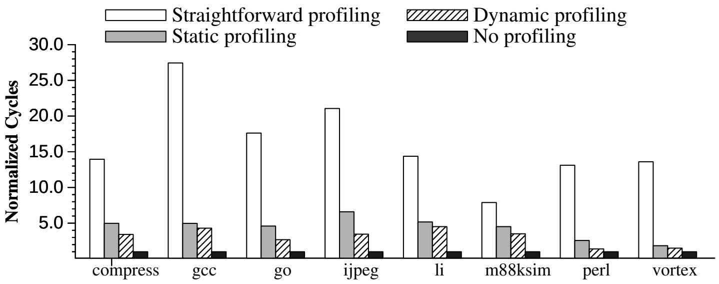

Fig. 3. Cycles used in benchmarks

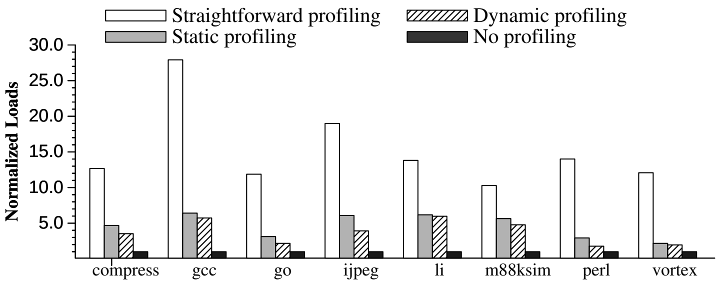

Fig. 4. Load Instructions Executed

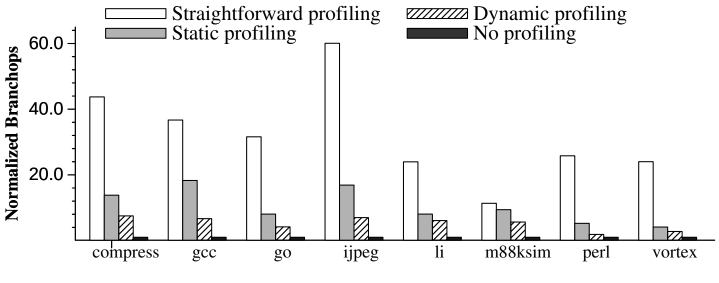

Fig. 5. Branch Instructions Executed

5.4 Specialization Performance

This paper is focused on improving the performance of value profiling. Our system automates both the selection of profiling points for profiling and the selection of value profiles for specialization. Figure 6 shows the speedup of the specialized program over running `alto` with all optimizations other than specialization. As Figure 6 shows, specialization yeilds speedups of up to 14.1% on significant programs such as the SPEC95 integer benchmarks. See [15] for further discussion of specialization performance in `alto`.

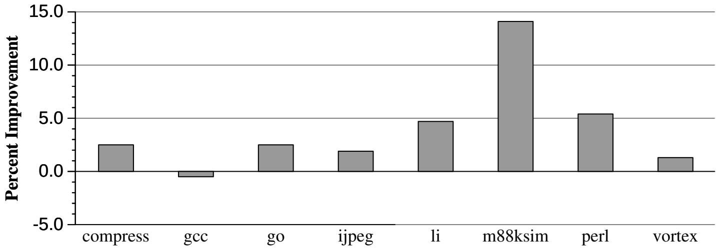

Fig. 6. Specialization Performance

6 Related Work

This work is closely related to the value profiling work of Calder *et al.*[5]. Their implementation supports profiling of both memory locations and registers. They report performance numbers for profiling the register result of every load instruction. Our current implementation profiles only registers. The mechanics used in our implementation are very similar to what Calder *et al.* refer to as "full" profiling.

Calder *et al.* also use convergent profiling to reduce the overhead of profiling in their system. Convergent profiling uses an invariance variable to determine if profiling is needed for a particular load instruction. This invariance variable is updated each time the load instruction is reached. If the table is not changing, profiling will be turned off for a while, and later turned back on for more profiling. An important invariant for us was that the count of a value in the value profile was a lower bound on the number of times the value actually appeared at run time. At the same time, we would like the count to be as close to accurate as possible. The higher the frequency of a value, the more benefit that will accrue for that program point when it is considered for optimization. Convergent profiling loses information that "full" profiling retains. Calder *et al.* report an average slowdown of 10x for convergent profiling. We chose to use a cost-benefit analysis to reduce the number of program points profiled, rather than lose information through convergent profiling. Despite using "full" profiling, our overheads are considerably lower than those reported by Calder *et al.*

Also related is work on runtime code specialization, which uses a semi-invariant variable, along with its value to simplify the the code in the common case. Existing techniques require the programmer to annotate the code to identify candidates for code specialization [1, 8, 9] Value profiling can then be used to determine which of the programmer-specified variables exhibit predictable behavior at runtime. Our implementation automates this process, identifying candidates for profiling, and then choosing among these candidates for optimization.

There is a large body of work relating to value prediction and speculation, e.g., see [11, 12] . This involves using a combination of compiler generated information and runtime information. For example, a load instruction will often load the same value as it loaded the previous time it was executed. Predictable instructions such as these can be speculatively executed, and then checked at a later time to make sure that execution was correct. If the execution was incorrect, then recovery code must be executed.

Value profiling can help to classify predictable instructions for speculative execution. Value profiling could also indicate which of several predictors will best suit a particular instruction, increasing prediction accuracy.

7 Conclusion

Value-based optimizations are becoming increasingly important for modern compilers [9] [8]. Performing these optimizations relies on information gathered at runtime. Knowledge of how the optimizer will use this information can make the profiler much more efficient. This paper describes our techniques for reducing the cost of gathering value profiles, both in time and space, by tightly integrating the value profiler with the optimizer that uses the value profiles.

Our optimizer makes use of a careful cost-benefit analysis to choose optimization opportunities. We use this same analysis to select only those $\langle$ program point, register $\rangle$ pairs for profiling that could be selected for optimization. This results in less than 1% of all program points being instrumented. Our profiler also makes use of this knowledge to modify its behavior while running, stopping profiling for program points that are no longer profitable. Our profiler will usually not instrument an instruction in the middle of a heavily executed loop, since the cost-benefit analysis can predict that such an instruction will not be chosen for optimization.

Our techniques result in a slowdown of 1.5-4.5x over the original code, down from a 10-28x slowdown for a straightforward approach. We also reduce the space requirements for a value profile to an average of 5% of that required by a straightforward approach.

References

1. J. Auslander, M Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effectvie dynamic compilation. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, May 1996.
2. T. Ball and J. R. Larus, "Optimally Profiling and Tracing Programs", *Proc. 19th. Symposium on Principles of Programming Languages*, Jan. 1992.

3. T. Ball and J. R. Larus, “Efficient Path Profiling”, *Proc. MICRO-29*, Dec. 1996.
4. B. Calder, P. Feller, and A. Eustace. Value profiling. In *30th Annual International Symposium on Microarchitecture*, pages 259–269, December 1997.
5. B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. In *Journal of Instruction Level Parallelism*, 1999.
6. P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, “Profile-guided automatic inline expansion for C programs”, *Software Practice and Experience* vol. 22 no. 5, May 1992, pp. 349–369.
7. R. Cohn and P. G. Lowney, “Hot Cold Optimization of Large Windows/NT Applications”, *Proc. MICRO29*, Dec. 1996.
8. C. Consel and F. Noel. A general approach to run-time specialization and its application to c. In *23rd Annual ACM Symposium on Principles of Programming Languages*, pages 145–146, Jan 1996.
9. D. Engler, W. Hsieh, and M. Kaashoek. ‘c: A language for high-level, efficient, and machine-independent dynamic code generation. In *23rd Annual ACM Symposium on Principles of Programming Languages*, pages 131–144, Jan 1996.
10. P. Feller. Value profling for instructions and memory locations. Master’s thesis, UCSD, 1998.
11. C. Fu, M. Jennings, S. Larin, and T. Conte. Software-only value speculation scheduling. Technical report, Department of Electrical and Computer Engineering, North Carolina State University, June 1998.
12. F. Gabbay and A. Mendelson. Can program profiling support value prediction? In *Proceedings of the 30 th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 270–280, Dec 1997.
13. A. J. Goldberg, “Reducing Overhead in Counter-Based Execution Profiling”, Technical Report CSL-TR-91-495, Computer Systems Lab., Stanford University, Oct. 1991.
14. S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufman, 1997.
15. Robert Muth, Scott Watterson, and Saumya Debray. Code specialization using value profiles. In *Static Analysis Symposium*, July 2000, pp. 340-359.
16. R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere, “`alto`: A Link-Time Optimizer for the Compaq Alpha”, *Software—Practice and Experience*, vol. 31 no. 1, Jan 2001, pp. 67-101.
17. K. Pettis and R. C. Hansen, “Profile-Guided Code Positioning”, *Proc. SIGPLAN ’90 Conference on Programming Language Design and Implementation*, June 1990, pp. 16–27.
18. B. Calder and D. Grunwald, “Reducing Indirect Function Call Overhead in C++ Programs”, *Proc. 21st ACM Symposium on Principles of Programming Languages*, Jan. 1994, pp. 397–408.
19. U. Hölzle and D. Ungar, “Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback”, *Proc. SIGPLAN ’94 Conference on Programming Language Design and Implementation* (PLDI), June 1994, pp. 326–336.