Code Specialization based on Value Profiles *

Robert Muth Scott Watterson Saumya Debray

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

Abstract. It is often the case at runtime that variables and registers in programs are "quasi-invariant," i.e., the distribution of the values they take on is very skewed, with a small number of values occurring most of the time. Knowledge of such frequently occurring values can be exploited by a compiler to generate code that optimizes for the common cases without sacrificing the ability to handle the general case. The idea can be generalized to the notion of *expression profiles*, which profile the runtime values of arbitrary expressions and can permit optimizations that may not be possible using simple value profiles. Since this involves the introduction of runtime tests, a careful cost-benefit analysis is necessary to make sure that the benefits from executing the code specialized for the common values outweigh the cost of testing for these values. This paper describes a static cost-benefit analysis that allows us to discover when such specialization is profitable. Experimental results, using such an analysis and an implementation of low-level code specialization based on value and expression profiles within a link-time code optimizer, are given to validate our approach.

1 Introduction

Knowledge that an expression in a program can be guaranteed to evaluate to some particular constant at compile time can be profitably exploited by compilers via the optimization known as constant folding [17]. 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. A similar situation holds in partial evaluation, where a variable has to be static in order to permit specialization [15]. In practice, it is often the case that an expression at a point in a program "almost always" takes on a particular value [6]. As an example, in the SPEC-95 benchmark *perl*, the function *memmove* is called close to 24 million times. The argument giving the size of the memory region to be processed has the value 1 in 70% of these calls. 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

* This work was supported in part by the National Science Foundation under grants CDA-9500991, CCR-9711166, and ASC-9720738.

advantage of this information to improve the performance of the program. Information about the relative frequency of occurrence is given by *value profiles*: a value profile for a variable or register x at a program point p is a (partial) probability distribution on the values taken on by x when control reaches p during program execution. This idea can be generalized to the notion of *expression profiles*, which profile the runtime values of arbitrary expressions and can permit optimizations that may not be possible using simple value profiles. Unfortunately, classical compiler techniques cannot take advantage of knowledge of the distribution of values, and optimize for the common case, in situations where a variable may take on multiple values at runtime. The idea behind value-profile-based code specialization is to allow such optimization.

From a semantic perspective, the transformation we use is very simple. To specialize a code fragment C for a value v of a register r,[1] we simply replace C by the equivalent code '**if** $(r == v)$ **then** C **else** C.' Once this has been done, "ordinary" specializing and optimizing transformations suffice to specialize the *true*-branch of this conditional to the value v of r. The resulting code has 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 runtime test '**if** $(r == v)$ ...' is required since we cannot guarantee that r will take on only the value v at that point. This idea can be generalized to multiple values: given a probability distribution on these values, we can use a collection of tests such as that above, organized as an optimal binary search tree, to choose between the specialized versions. For simplicity of discussion, we focus on specialization for a single value in this paper, since this illustrates the technical issues that arise.

Notice that this transformation is obviously semantics-preserving, can be applied anywhere, to any variable or register and any value (subject to any applicable type constraints), without requiring, for example, a binding-time analysis. This is the primary strength of our approach, and it allows optimizations that would not be possible otherwise; it is also our biggest weakness, because we have so little to guide us in exercising the tremendous freedom that we are given. For example, notice that due to the runtime test that has been introduced, the code resulting from specialization, shown above, is actually less efficient than the original for values of `r` other than v. Thus, value-profile-based specialization reduces cost of some execution paths, but the cost of other paths increases. If this tradeoff is not assessed carefully, it can result in significant performance degradation. In general, the technical issues that have to be addressed during value-profile-based code specialization are as follows:

1. we have to determine the program point[2] p where the specialization should begin (this corresponds to the point where runtime tests on values have to be inserted, as discussed above);

[1] In general, specialization can be carried out based on the value of a register, variable, or memory location, or relationships between such values. To simplify the discussion, and because our current implementation carries out specialization based on register values, we refer to register values when discussing specialization.

[2] For our purposes a "program point" refers to the points immediately before or after an instruction; this includes the entry and exit points of basic blocks.

2. we have to identify the register r whose values we are interested in, and the particular value(s) v of this register that we specialize for;
3. we have to determine the actual code fragment C that is to be subjected to specialization.

The primary contribution of this paper is a low-level static cost-benefit analysis that allows us to evaluate the runtime tradeoff mentioned above—where specialization can reduce the runtime cost of some execution paths but increase the cost of others—and guide the specialization process. This analysis is crucial, since specializing a piece of code for too many different values, or specializing code where the benefits of specialization are not high enough, can lead to a performance degradation. We then describe details of how the analysis, specialization, and subsequent code optimization have been automated and integrated into a link-time code optimizer (*alto*), and give experimental results to validate our ideas.

2 Code Specialization

Value-profile-based code specialization is a three-step process:

1. identify program points and registers where specialization may be profitable using basic block profiles;
2. obtain value and expression profiles for those program points;
3. use these profiles to carry out specialization for those program points where this is deemed profitable

A *specialization triple* is a triple of the form (p,r,v), where p is a program point, r is a register, and v is a value for that register. These triples identify the runtime tests that have to be inserted in the context of value-profile-based specialization and the program points where they must be inserted. The *specialization region* of a triple (p,r,v) refers to the region of code that is chosen for specialization; this identifies the code fragments that appear in the **then**- and **else**-branches of the runtime test corresponding to that triple.

Section 2.1 describes a benefit analysis that is fundamental to our approach. In Sections 2.2 through 2.4 we discuss the three steps mentioned above. Section 2.5 provides an example illustrating our approach.

2.1 Estimating Benefits of Specialization

Our value profiling and specialization decisions are guided by estimates of the benefit that would be obtained from code specialization given the knowledge that the value of a register r is known at a program point p. This estimate is denoted by $\mathsf{Benefit}(p,r)$. There are two components to the computation of benefits:

(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 r of an instruction I allows us to determine the value computed by I. In this case, I is said to be *evaluable* given r, written $\mathsf{Evaluable}(I,r)$. If I is evaluable given r, the benefits obtained from specializing other instructions that use the value computed by I for a particular value of r can also be 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 $\mathsf{IndirBenefit}(I,r)$.

If we know the values of all operands to an instruction, we can compute the result v of the instruction, and propagate this value to all instructions that use v. There is therefore no need to execute this instruction at run-time. The savings obtained from knowing the operand values for an individual instruction is essentially the latency of that instruction (i.e., the number of cycles it takes to execute), if knowing the operand values allows us to determine the value computed by that instruction, and thereby eliminate that instruction entirely[3] (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:

$$\begin{aligned}
\mathsf{Benefit}(p,r) &= \sum_{I \in \mathsf{Uses}(p,r)} (\mathsf{Freq}(I)^4 \times \mathsf{Savings}(I,r) + \mathsf{IndirBenefit}(I,r)) \\
\mathsf{IndirBenefit}(I,r) &= \textbf{if } \mathsf{Evaluable}(I,r) \textbf{ then } \mathsf{Benefit}(p', \mathit{ResultReg}(I)) \textbf{ else } 0.
\end{aligned}$$

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

These equations for computing benefits propagate information from the uses of a register to its definitions. They can be recursive in general, corresponding to a cycle in the use-definition chain. The usual approach to solving recursive equations in the context of program analysis is to use an iterative fixpoint computation (e.g., see [9]). In our case, however, it is not obvious from a pragmatic standpoint that this is the right thing to do. The reason for this is that propagating benefit information around a cycle is meaningful only if we know, *a priori*, that the loop will be unrolled later (otherwise we cannot specialize the loop body for values encountered on different iterations of the loop). When carrying out loop unrolling, however, it is essential to take into account machine-level resources such as registers and the instruction cache: excessive unrolling that does not consider these factors can result in severe performance degradation (e.g.,

[3] 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 do not pursue the details here due to space constraints.

[4] $\mathsf{Freq}(I)$ refers to the dynamic execution frequency of the instruction.

see [11]). For this reason, the decision as to whether the loop should actually be unrolled is not made at the time of the cost-benefit computation, but later, based in part on information obtained from value and expression profiling (see Section 3). If benefit information is propagated around the loop but the loop subsequently is not unrolled (e.g., due to cache considerations), we can get wildly optimistic benefit values. These values can mislead the cost-benefit estimation and lead to the introduction of useless runtime tests, thereby degrading performance.

We therefore have a chicken-and-egg problem: propagating information around cycles when identifying candidates for value profiling requires knowledge of whether or not loops will be unrolled; but the decision of whether or not to unroll a loop depends upon, among other things, knowledge of value profiles. As a practical matter, it happens that complex low-level analyses of machine code programs (as in our implementation) and determination of value profiles are both quite expensive; this greatly limits our choices in dealing with this circular dependence. The approach we take, therefore, is one where we attempt to "do no harm:" we conservatively assume that loops will not be unrolled when carrying out our benefit analysis, and therefore do not propagate information along loop back edges. This has the drawback that it can sometimes cause us to underestimate the benefit that might actually have been obtained if cycles had been taken into account; as a result we could miss some opportunities for optimization. Note, however, that this is conservative, in the sense that it will not insert runtime tests or specialize code that is not worth specializing.

Our approach, therefore, is to obtain approximate solutions to the benefit equations given above, where the approximation occurs in the handling of loops as discussed above. This is done as follows. First, let the *defining instruction* of an instruction I, written $\mathit{defInst}(I)$, be the (single) instruction J such that knowing the value computed by J into its destination register allows us to determine the value computed by I; if there is not a single such instruction, the defining instruction is undefined, denoted by $\perp$.[5] Use-definition chains are used to compute the defining instruction for an instruction $I \equiv$ '$r_c = r_a \oplus r_b$' as follows:

(i) if the values of both r_a and r_b are statically known, $\mathit{defInst}(I) = \perp$;

(ii) otherwise, if the value of one of the operand registers is statically known, and there is a single definition J for the other operand register that reaches I, then $\mathit{defInst}(I) = J$;

(iii) otherwise, if $r_a = r_b$ and there is a single definition J for r_a that reaches I, then $\mathit{defInst}(I) = J$;

(iv) otherwise $\mathit{defInst}(I) = \perp$.

In case (i), all of the operands of an instruction I are known statically. This instruction will be specialized without relying on value profiles at all. For the purpose of value profile based specialization, therefore, we do not consider such instructions. A convenient way to do this is by setting $\mathit{defInst}(I)$ to $\perp$. In case (iv), neither of the operands of an instruction are known statically. We do not wish to propagate benefit from case (iv)

[5] Our implementation introduces, at the entry to each basic block that has more than one predecessor, a pseudo-instruction, similar to a SSA ϕ-function, that defines each register that is live at that point and has more than one definition reaching it. The notion of defining instructions extends to such pseudo-instructions in the obvious way.

instructions since they cannot be evaluated after knowing the value of a single defining instruction.

The benefit for each instruction can now be computed as follows. Let *Benefit*(I), where I is an instruction, denote the value $\mathsf{Benefit}(p,r)$, where p is the program point immediately after I and r is the destination register of I. First, we mark all instructions in the program as *unprocessed*, and set *Benefit*$(I) = 0$ for each instruction I. The following is then repeated until no new instruction can be marked as *processed*:

```
for each unprocessed instruction I do
    /* memory operations are not specialized away */
    if I is not a memory operation then
        J = defInst(I);
        if J ≠ ⊥ and all instructions dependent on I have been processed then
            Benefit(J) += Benefit(I) + Savings(I,r),
                where r is the destination register of J;
            mark I as processed;
        fi
    fi
od
```

This algorithm will not process any instruction that is involved in such a cycle, since *Benefit*(I) is added to *Benefit*(J) only after all of the instructions dependent on I have been processed, i.e., after the value of *Benefit*(I) has stabilized. This will cause benefit information to not be propagated around loops, for the reasons discussed above. An added benefit of such an approach is that of efficiency: disallowing information propagation around cycles makes the code for estimating benefits simpler and faster.

2.2 Identifying Candidates for Specialization

In order to reduce the time and space overheads for value profiling as far as possible, we attempt to identify candidate (*program point*, *register*) pairs for which specialization could conceivably yield a performance improvement if we had a sufficiently skewed runtime distribution of values. Once the benefits associated with each instruction have been computed as described above, we only consider those instructions whose benefit is equivalent to the elimination of at least a single instruction from a "hot" basic block. The intent is to avoid the overheads associated with value profiling, and perhaps specializing, instructions where this is unlikely to lead to a noticeable improvement in performance. Notice that this does not mean that instructions considered for specialization must actually cause the elimination of instructions in hot basic blocks, but simply that the savings incurred from specialization be large enough to be comparable to the elimination of at least one instruction from a hot block. Employing this cost-benefit analysis reduces the overhead of profiling significantly. We discuss this in more detail in Section 4

Alto uses a two-stage profiling scheme where basic block profiles are first generated, and these are used to determine which value profiles to compute. At this point, therefore, we have basic block execution counts. To determine the basic blocks that are "hot," i.e., executed sufficiently frequently, we start with a value ϕ in the interval (0,1] and determine the largest execution frequency threshold N such that the set of basic blocks that have execution frequencies exceeding N together account for at least the fraction ϕ of the total number of instructions executed by the program (as indicated by its basic block execution profile). For the purposes of value-profile-based specialization, we use an empirically derived value of $\phi = 0.50$, i.e., the hot basic blocks consist of those that allow us to account for at least 50% of the instructions executed at runtime.

2.3 Value Profiling

Given a set of (*program point*, *register*) pairs to be value-profiled, we use a scheme based on that of Calder *et al.* [6] for obtaining value profiles. As mentioned earlier, our implementation of value profiling obtains profiles for registers only, not for memory locations. The actual profiling is carried out by a function created for this purpose. This function, which is added to the program as part of the instrumentation code and invoked at the profiling points, compares the value of the register in question with the contents of a fixed-size table of previously encountered values. If the current value is already in the table, the count of that value is incremented. Otherwise, if the table is not full, the value is added to the table and its count initialized to 1. If the table is full the value is ignored. Periodically, the table is cleaned by evicting the least frequently used values from the table: this allows new values to enter the table. We also keep track of the total number of times execution passes through the point p by incrementing a counter associated with that point.

2.4 Carrying out the Specialization

Code specialization involves two steps: (1) identification of the particular specialization triples, and the corresponding specialization regions, that should be specialized; and (2) transforming the program appropriately.

The benefit computation described in Section 2.1 is used to identify the specialization triples for which code specialization is worthwhile. Once the actual value profile has been obtained, we know the distribution of the values taken on at the points that have been profiled and can determine the probability $\mathsf{prob}(v)$ with which a value v occurs. The benefits of specialization have to be weighed against the costs incurred due to runtime tests. The cost of such a test depends on the register and value being tested: e.g., testing for the value 0 is usually fairly cheap, while testing for a non-zero floating point constant may incur a load from memory. The cost of testing whether a register r has a value v is denoted by $\mathsf{TestCost}(r,v)$. Specializing at a program point p for a value v of a register r is then worthwhile only if the marginal benefit, given by

$$\mathsf{Benefit}(p,r) \times \mathsf{prob}(v) - \mathsf{TestCost}(r,v) \times \mathsf{Freq}(p),$$

is equivalent to at least one hot instruction (cf. the discussion in Section 2.2).

Once we have identified the set of specialization triples for which specialization is worthwhile, we have to choose which of these should actually be specialized. An issue that must be addressed here is that the specialization regions for different such triples may overlap. This is illustrated by the following instruction sequence:

```
ld  r5, 0(r4)            # r5 := load from 0(r4)
and r5, 0xff, r6         # r6 := r5 & 0xff
```

Suppose that we have value profiled register `r5` after the `ld` instruction and register `r6` after the `and` instruction, and that based on the cost benefit analysis, both of these instructions are candidates for specialization. However, the program points are dependent—`r6` is computed from `r5`—and their specialization regions overlap. Depending on the circumstances, it might be better to specialize based on the `ld` instruction because more instructions use the result of this instruction; in other situations, it might be better to specialize based on the `and` instruction because its value distribution might be more skewed. In such cases, we specialize only the more promising one, based on the cost benefit analysis; in the case of a tie, the program point that dominates the other is chosen (as discussed below, overlaps are not possible unless one of the points dominates the other).

Given a set of specialization triples, we have to determine the specialization region associated with each of them. The basic intuition is that given a triple (p,r,v), we want to identify the instructions that, directly or indirectly, use the value of r available at p, and so might potentially benefit from specialization. We first make precise the notion of an instruction using a value "directly or indirectly." Given a program point p and register r, we say that (p,r) *influences* an instruction I if (i) I uses the value of r at p; or (ii) there is an instruction J at a program point p' such that: J defines a register r'; (p,r) influences J; and (p',r') influences I. Then, given a triple (p,r,v), the specialization region for this triple is defined to be the smallest set of basic blocks R such that

– R contains the basic block B_p containing p is in R;
– if (p,r) influences an instruction I occurring in a basic block B_I, and p dominates B_I, then B_I is in R; and
– if B is in R, $B \neq B_p$, and B' is a (immediate) intra-procedural predecessor of B in the control flow graph of the program, then B' is in R.

It is not hard to see that, given a specialization triple (p,r,v), the basic block B_p containing p dominates every block in the specialization region of this triple. This is necessary for correctness: we have to ensure that any execution path that can reach the specialization region of this triple must pass through the test inserted at p.

There are two issues that are not addressed by this definition of specialization regions. The first is that, given a triple (p,r,v), it may happen that (p,r) influences an instruction I but the basic block B_I containing I is not in the specialization region of this triple because p does not dominate B_I. This problem can be remedied by duplicating code so as to make p dominate B_I. This is an issue that is, by and large, orthogonal to the main focus of this paper, and so is not pursued further here. The second is that, as given, this definition does not take into account the size of a specialization region relative to the benefits obtained from its specialization. It may happen that an instruction I in a block B_I that is very far away from the point p is influenced by the value of a register r at p. If we include B_I in the specialization region, it is necessary to also include all of the blocks between p and B_I, even though these blocks may not benefit from specialization. This could, in extreme cases, give rise to large specialization regions in order to include distant influenced instructions. This can be handled using a notion of *density* of influenced instructions, analogous to the notion of density of case labels used for code generation for `switch` statements [5], to limit the specialization regions to code that contains a sufficiently high proportion of instructions that would benefit from the specialization. Our current implementation does not address this issue.

The final step is to actually carry out the code transformations for specialization. The transformations that are effected during specialization can be quite involved. Since much of this functionality is already available elsewhere in our system in the routines that implement various analyses and optimizations, we attempt to have as little code as possible for transformations specifically geared towards value-profile-based specialization. Our goal is to transform the code just enough, at this point, that the desired specialization will subsequently happen in the course of "ordinary" optimizations. We have only two transformations specific to value-profile-based specialization:

1. The basic transformation, aimed at transferring control to specialized code when a register has the appropriate value, is implemented as follows. When specializing for a triple (p,v,r), we simply create a copy C' of the specialization region C for that triple and insert a test at program point p that tests r and branches to the copy if r's value is v.
2. When value profiling indicates that the iteration count of a (hot) loop C has a sufficiently skewed distribution, we may generate a specialized version C' of that loop that has been unrolled some number of times. The specific number of unrollings is based on the sizes of the bodies of the loop under consideration as well as those of any loops in which it is nested, together with the size of the instruction cache, so as to avoid excessive unrolling that could adversely affect the i-cache utilization of the program (e.g., see [11]). Control is transferred to the unrolled loop by testing the register r controlling the number of iterations against a particular value v, as for the basic transformation above.

Once the code has been transformed as described, the information that r has the value v when control reaches the cloned region C', but not the original code fragment C, is propagated during the course of conditional constant propagation [19]; The actual specialization of the code then takes place in the course of normal optimizations, which exploit the additional information that is available about the value of r—and, possibly, other computations that use the value of r—to effect a variety of optimizing transformations. Using this approach we are able to reuse much of the optimization infrastructure of our system for value-profile-based specialization, leading to a simpler system that is easier to implement, debug, and maintain.

Given a specialization triple (p,r,v), a variety of idioms may be used to implement the test inserted at the program point p, depending on the magnitude of the value v and whether or not there is a free register available. If a free register r' is available, we simply compute the difference of r and v into r', then conditionally branch to the cloned code if r' is zero. If there are no free registers available, if v is small enough to be an immediate operand the following pair of instructions is inserted:

```
subq r, v, r # r := r - v
beq r, B_clone  # if (r = 0) goto B_clone;
                # else fall through to original code
```

To compensate for the effect of the `subq` instruction, we add the instruction '`addq r, v, r`' at the entry to both the original specialization region and its clone. If v is too big to be an immediate operand, one or more instructions may be needed to compute it into a register; however, the cost of doing so will have been taken into account in $\mathsf{TestCost}(r,v)$.

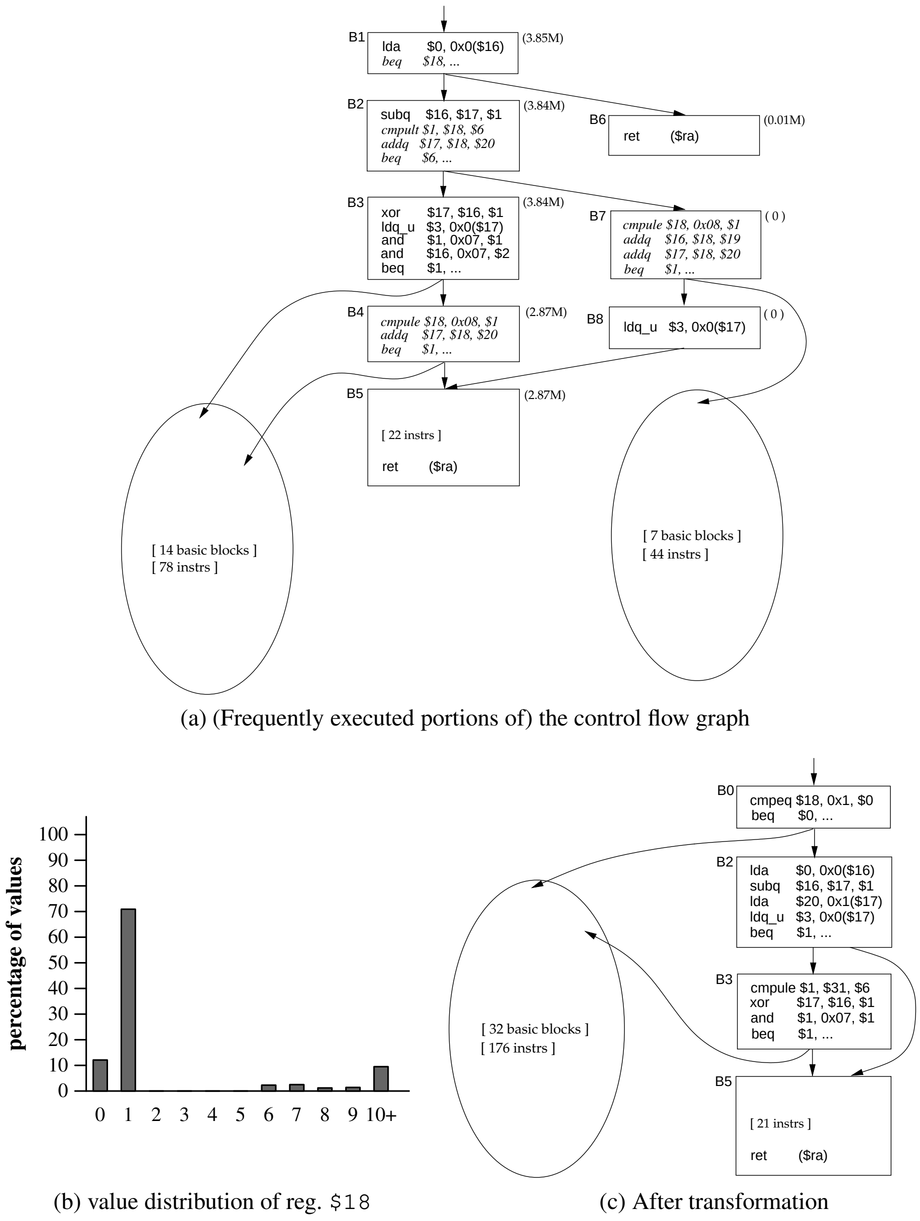
2.5 An Example

As an example illustrating our approach, we consider the function *memmove()*, from the SPEC-95 benchmark *perl*. The frequently executed portion of its control flow graph is shown in Figure 1(a), with the execution count of each basic block shown in parentheses on the right of the block. Instructions that (directly or indirectly) use the value of the third argument, passed in register `$18`, are shown in italics. The distribution of values for this register is shown in Figure 1(b): notice that over 70% of the time, this register has the value 1.[6] The instructions along the critical path of the function that are influenced by the value of register `$18` are shown in italics in Figure 1. We focus on the transformations that occur along the critical path of this function in the course of specialization, since these have the largest impact on performance:

- [+2 instructions] The most commonly occurring value for this register is 1, and value-profile-based specialization introduces a test for this value in block B0 (see Figure 1(c)).
- [−3 instructions] Constant propagation then causes the elimination of the following instructions: '`beq $18, ...`' from block B1; and the pair '`cmpule $1, 0x08, $1`', '`beq $1, ...`' from block B4 (a similar pair is eliminated from block B7).
- [−2 instructions] The elimination of the '`beq $1, ...`' instruction from block B4 causes the deletion of the control flow edge out of B4 away from the critical path (i.e., into the oval marked "[14 basic blocks]"). This has two effects. First, it causes register `$20` to become dead at the end of block B4, which allows the deletion of the instruction '`addq $17, $18, $20`' in B4. Second, it causes the instruction '`and $16, 0x07, $2`' to become partially dead in block B3; partial dead code elimination then moves it out of B3, and hence out of the critical path.
- [−1 instruction] For the instruction pair '`cmpult $1, $18, $6`', '`beq $6, ...`' in block B2, given that the value of register `$18` is 1, the instruction '`cmpult $1, $18, $6`', which does an unsigned comparison of registers `$1` and `$18`, will yield a value of 1 only if register `$1` is 0. The optimizer recognizes this and replaces this pair of instructions by the single instruction '`beq $1, ...`'.
- [−1 instruction] Constant propagation of the value of register `$18` also succeeds in deleting a `mskql` instruction (a bit mask instruction used in byte manipulations) from block B5.

The resulting code is also subjected to other transformations, such as code hoisting and basic block fusion, that are enabled by the transformations described above. The resulting code is shown in Figure 1(c). The overall effect of these transformations is to reduce the length of the critical path through this function from 37 instructions to 32 instructions, a reduction of 13.5%.

[6] The basic block execution counts given in Figure 1(a), as well as the value distribution shown in Figure 1(b), correspond to the training inputs of the SPEC-95 benchmarks, since that is what a compiler would use to reason about the program. Those mentioned in Section 1 refer to the SPEC reference inputs.



(a) (Frequently executed portions of) the control flow graph

(b) value distribution of reg. `$18`

(c) After transformation

Fig. 1. A specialization example: the function *memmove()* (from the SPEC-95 benchmark *perl*)

3 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 "is the value of register r_a different from that of register r_b?" In general, as shown below, 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 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*.

3.1 Loop Unrolling

Here we try to determine the distribution of the number of iterations of the loop. In simple cases, this may be just the value of a variable: e.g., in a loop of the form

```
for (i = n; i > 0; i--) { ... }
```

In general, however, the iteration count may depend on more complex expressions whose value may not be known at compile time: e.g., in a loop of the form

```
for (i = m; i < n; i++) { ... }
```

iteration count is given by the expression `n-m`. This expression does not appear in the source code. If the iteration count of a loop can be predicted given the value of an expression prior to the execution of the loop, and this distribution is sufficiently skewed, we may choose to generate, subject to i-cache considerations, an unrolled version of the loop based on that information. Notice that the test to decide whether or not to execute the unrolled version of a loop is made by a single test that is outside the loop, so the associated overhead is not very high.

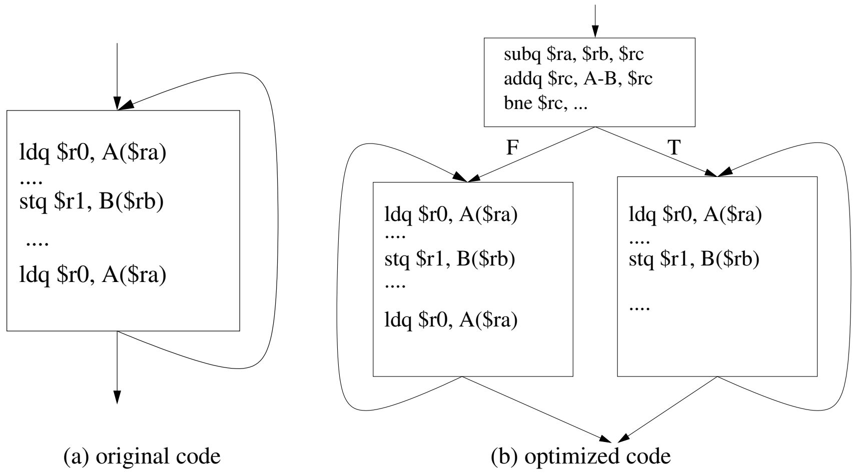


Fig. 2. Load Avoidance Example

3.2 Load Avoidance

The goal here is to use expression profiling to determine relationships between memory access operations, and thereby avoid unnecessary memory operations where possible. Suppose we have a sequence of operations (typically within a loop) as shown in Figure 2(a). Let $k\,(r)$ represent the address obtained by adding k to the contents of register r. If we can guarantee that the addresses $A\,(r_a)$ and $B\,(r_b)$ will never overlap, we can eliminate the second load operation in the sequence shown. However, in practice, it is very difficult to prove that the two instructions will never overlap. We use expression profiling to determine how often the two instructions overlap at runtime, and use this information to optimize the code.

We first identify the instructions that define the index registers r_a and r_b and attempt to determine the rate at which these registers change within the loop; if either register is defined by a load operation from a fixed location, we attempt to determine the rate at which the value at that location changes. If we can obtain constant rates of change δ_a and δ_b for these registers, respectively, we consider the following cases:

($\delta_a = \delta_b$): Here, it suffices to test whether $A\,(r_a) \neq B\,(r_b)$ at entry to the loop; the expression profiled in this case is essentially this expression, simplified as far as possible to reduce runtime overheads.

($\delta_a \neq \delta_b$): Assume that $\delta_a > \delta_b$ and both rates are non-negative (the other cases are analogous). There is no conflict between the two addresses if, at entry to the loop, either $m_a(r_a) > m_b(r_b)$ or $(m_a(r_a) + n \times \delta_a) < m_b(r_b)$, where n is the iteration count of the loop. In this case we profile these two expressions separately.

In our example, r_a and r_b are unchanged within the loop. Therefore, we profile the expression $A\,(r_a) \neq B\,(r_b)$. If expression profiling determines that at runtime the above expression is true sufficiently frequently, we optimize the code. The specialized code from our example is shown in Figure 2(b). Again, in the specialized code the expression is tested once outside the loop and so is not very expensive. Note that the aliasing test is not present in either the source code or the original executable.

3.3 Transformation

Expression-profile-based code transformations are nearly identical to those performed for value-profile-based code specialization. A clone of the affected blocks is created, and a test is inserted to choose between the specialized code and the original code. Additionally, information about (non-)aliasing between pointers, obtained from expression profiling, is attached to the relevant basic blocks. We then rely on other parts of our system to eliminate the unnecessary load and store instructions.

As an example of the application of expression profiling, in the SPEC-95 benchmark *m88ksim*, expression profiling allows us to determine that three pointers in a heavily executed loop within the function *alignd* are usually not aliased; this information is used to eliminate several redundant memory accesses and thereby effect a significant speed improvement.

4 Experimental Results

Program	*Execution Time* (secs)		T_{spec}/T_{nospec}
	unspecialized (T_{nospec})	specialized (T_{spec})	
compress	260.75±0.02%	254.25±0.30%	0.975
gcc	220.45±0.16%	221.58±0.08%	1.005
go	309.43±0.81%	301.57±0.26%	0.975
ijpeg	327.24±0.02%	320.95±0.41%	0.981
li	249.59±0.03%	237.97±0.04%	0.953
m88ksim	220.21±0.08%	189.19±0.06%	0.859
perl	178.96±1.91%	169.54±0.51%	0.947
vortex	301.22±1.09%	297.35±0.05%	0.987

Table 1. Impact of Value-Profile-based Specialization on Execution Time

We have implemented the ideas described here within the *alto* link-time optimizer [18]. The programs used were the 8 integer programs from the SPEC-95 benchmark suite. The programs were compiled with the vendor-supplied C compiler V5.2-036, invoked as `cc -O4`, with additional linker options to retain relocation information and produce statically linked executables.[7] Both the initial execution frequency profiles as well as the value profiles for each program were obtained using the SPEC training inputs; the execution times reported were then obtained using the SPEC reference inputs.

The results of our experiments are shown in Table 1. The second column of this table, with heading "unspecialized", gives the execution time for the executables using all optimizations within *alto* except for value-profile-based specialization, while

[7] We use statically linked executables because *alto* relies on the presence of relocation information for its control flow analysis. The Digital Unix linker `ld` refuses to retain relocation information for non-statically-linked executables.

the third column, with heading "specialized", gives the execution times when value-profile-based specialization is carried out as well. The last column gives the ratio of the execution times with and without specialization. The timings were obtained on a lightly loaded DEC Alpha workstation (i.e., with no other active processes) with a 300 MHz Alpha 21164 processor with a split primary cache (8 Kbytes each of instruction and data cache), 96 Kbytes of on-chip secondary cache, 2 Mbytes of off-chip backup cache, and 512 Mbytes of main memory, running Digital Unix 4.0. In each case, we ran the program 10 times and discarded the biggest and smallest execution times; for the remaining runs, we computed the mean as well as the maximum deviation of any run from the mean. Our results are given in Table 1, with the maximum deviation expressed as a percentage of the mean.

It can be seen from these numbers that automatic value-profile-based specialization can yield noticeable performance improvements for nontrivial programs. Most of our benchmarks experience speedups, with *m88ksim* and *perl* experiencing the largest speedups of 14.1% and 5.6% respectively. Due to space constraints, a description of the reasons for the performance improvements in the various benchmarks is relegated to Appendix A. We have not yet determined the reasons for the slowdown in the *gcc* benchmark: sometimes, as shown here, the specialized code is slower than the unspecialized code, while at other times the specialized code is faster; we are currently investigating this problem. A detailed examination of the low-level performance of the specialized programs, using hardware performance counters, indicates that the performance of the specialized programs suffers from deficiencies in other parts of our system that we believe will not be difficult to rectify. For example, several of the specialized benchmarks suffer from an increase in mispredicted branches (*compress* by about 7%, *perl* by about 4%), which we suspect may be due to the layout of the code. The number of i-cache misses also goes up in some programs (*m88ksim* by 6%; *compress* by 16%, though in this case the miss rate is so low that it is not clear that this has a significant effect), again pointing to code layout as a possible culprit. We expect to be able to address these problems soon.

Program	*No. of Program Points*		
	Total	Profiled	Optimized
compress	16749	74	0+1
gcc	271899	7231	196+0
go	65328	1352	4+0
ijpeg	49650	243	5+1
li	32221	171	7+0
m88ksim	40867	253	16+0
perl	82462	501	14+0
vortex	113236	322	15+0

Table 2. Extent of Profiling and Specialization

Table 2 compares, for each benchmark, the total number of program points that could have been profiled/specialized (column 2) with the number that were actually profiled (column 3) and the number that were then optimized (column 4); the last of

Program	*Code Size* (Instructions)		I_{spec}/I_{nospec}
	unspecialized (I_{nospec})	specialized (I_{spec})	
compress	17381	17529	1.009
gcc	279429	281584	1.007
go	71046	71169	1.002
ijpeg	51045	52385	1.026
li	29106	29131	1.001
m88ksim	40865	41237	1.009
perl	82167	82304	1.002
vortex	103660	103743	1.001

Table 3. Impact of Value-Profile-based Specialization on Code Size

these entries are given in the form $m+n$, where m is the number of program points that were specialized and n the number of loops that were unrolled. This indicates that the our computation of the cost/benefit tradeoffs is highly selective: for most of the benchmarks fewer than 1% of the potential candidates for profiling are actually chosen for profiling (*gcc* comes in highest with a little under 2.5% of the candidates actually profiled). Table 3 shows, for each benchmark, the code growth that results from specialization. The small number of points chosen for profiling keeps the value profiling overhead under control, while of the small number of points chosen for specialization keeps the code growth modest. As mentioned previously in Section 2.2, our profiling overhead is considerably reduced by applying our benefit analysis before performing the value profiling. Calder *et al.* [7] report a 33x average slowdown for full value profiling on the SPEC-95 benchmarks. *Alto*, in contrast, produced 3x-9.5x slowdowns (6.3*x* on average) for value and expression profiling.

Figure 3 illustrates the overheads associated with value-profile-based specialization. It shows, relative to the time taken by *alto* to optimize an executable program without either value profiling or specialization, the following quantities: (i) the time taken to instrument the code for value profiling, i.e., to read in an executable file, identify candidates for value profiling, insert instrumentation code, and write out the instrumented binary (Section 2.2); and (ii) the time taken to specialize the program using value profiles, i.e., read in the program as well as the profile data, carry out all optimizations including value-profile-based specialization, and write out the optimized executable (Section 2.4). The initial cost-benefit computation to identify profiling candidates, together with the instrumentation overhead, results in overheads in the range of 20%–80% (about 44% on the average) compared to the time for ordinary processing by alto. Specialization based on value profiles incurs overheads of factors ranging from 1.6x to 2.1x (about 1.87x on the average).

5 Related work

There is a considerable body of work on program specialization within the partial evaluation community: Jones *et al.* give an extensive discussion and bibliography [15]. This work focuses largely on code specialization starting with known values for some or all

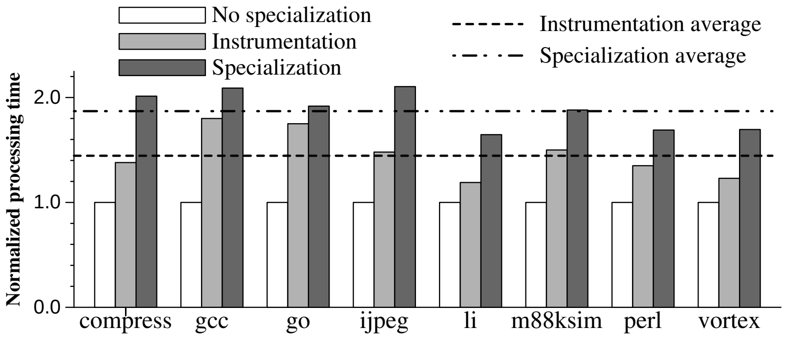


Fig. 3. Overhead of Value-Profile-Based Code Specialization

of a program's inputs. Specialization based on value profiles, where we reason about the runtime distribution of values taken on by a variable, is not considered.

In some ways, our approach to specialization is reminiscent of a transformation referred to as "the trick" in the partial evaluation literature (e.g., see [15]). There are two main differences between these transformations. The first is that "the trick" is applied to variables of *bounded static variation*, i.e., which take on values from a finite, statically known, set, while our approach does not have such a restriction (e.g., in the example discussed in Section 2.5, the variable that is specialized ranges over the set of integers). Furthermore, "the trick" offers no guidance regarding which values are worth specializing and which are not: because of this, automatic application of this transformation can be problematic if the candidate variable is of bounded static variation but ranges over a very large, albeit finite, set. The analysis we describe is intended to address precisely this problem. As such, it can be a useful complement to standard partial evaluation techniques.

Also related to our cost-benefit analysis is the work on speedup analysis in partial evaluation [2, 15]. This analysis starts with a binding-time annotated program, where variables whose values are statically known are marked as such. Speedup analysis estimates the asymptotic speedup that partial evaluation of the program would yield. By contrast to this work, we cannot assume that we have a binding-time annotated program—indeed, the whole point of our analysis is to take variables whose values cannot be statically predicted, and determine which if any, of the (possibly unboundedly many) values taken on by such variables might yield performance improvements. Another important difference is that we are concerned not with asymptotic speedups but rather with concrete improvements in speed, and therefore pay careful attention to low level issues such as the effects of specialization on instruction cache utilization (as discussed, for example, in Section 2.4 in the context of loop unrolling).

Some implementations of object-oriented languages attempt to mitigate the high cost of dynamically dispatched calls using a limited form of value-profile-based specialization. The idea, referred to as *type feedback* or *receiver class prediction* [1, 14], is to monitor the targets of dynamically dispatched function calls, and to use this information to inline the code for frequently called targets. The main limitation of this approach is that the specialization is restricted to dynamically dispatched function calls, and so will not be applied to "ordinary" code even if such code could benefit substantially from knowledge of the values most commonly encountered at runtime.

Calder *et al.* have investigated issues and techniques for value profiling [6]. Our implementation of value profiling was inspired by theirs and is very similar to it. While Calder *et al.* consider profiling both registers and memory locations, we only profile registers. We use a two-stage profiling process in order to reduce the time and space overheads. The idea is to first profile the application using a simple basic-block profiler such as pixie, and then use the execution frequency information so obtained to identify candidates for value profiling and specialization. In a different paper, Calder *et al.* discuss value-profile-based optimization [7]: they use hand-transformed examples to show that value-profile-based specialization can yield significant speed improvements. By contrast, our work describes value-profile-based specialization that is fully automatic and that has been integrated into a link-time optimizer.

Systems for dynamic code generation and optimization [4, 8, 12] are also confronted with tradeoffs between the cost of generating specialized code and the savings obtained from the execution of this code. The problem, while qualitatively similar to ours, is considerably more complicated in practice because the runtime costs include the cost of generating the specialized code, which can be difficult to estimate precisely. Systems that extend existing source languages with facilities for dynamic code generation, such as Tempo [8] and DyC [12], generally require users to annotate the program fragments that should be subjected to runtime code generation and specialization, effectively moving the burden of analyzing the cost-benefit tradeoff to them. Systems for dynamic optimization of conventionally optimized programs, such as Dynamo [4], rely on simple heuristics to determine whether a code fragment is worth optimizing: programs where these heuristics are inadequate can suffer noticeable performance degradation.

The work that is conceptually closest to that described here is some recent work towards automating the cost-benefit analysis for DyC [16]. The goals of this work are considerably more ambitious—and also more difficult—than ours. A direct comparison of the efficacy of the two systems is difficult, partly because they take very different approaches towards specialization (one is static, the other dynamic), and partly because the benchmarks used by the authors are mostly different from ours; of the benchmarks considered for DyC [13], the only one that is also considered by us is *m88ksim*. For this program, Grant *et al.* report an overall speedup of 5%, whereas we obtain a speedup of a little over 13%. Other studies by the authors of DyC suggest that, assuming that the cost-benefit tradeoff assessment can be made properly, runtime specialization can yield significant asymptotic speedups, albeit sometimes with fairly high break-even points [3].

6 Conclusions

This paper describes an implementation of low level code specialization based on value profiles. Fundamental to our approach is a low-level cost-benefit analysis that is used both to reduce the overheads due to value profiling and also to identify the code to be specialized. Experimental results indicate that the cost-benefit analysis is effective in filtering out unpromising candidates, and that several non-trivial programs experience noticeable performance improvements due to value-profile-based specialization.

Acknowledgements

We are grateful to Brad Calder for very helpful discussions as well as comments on an earlier version of this paper.

References

1. G. Aigner and U. Hölzle, "Eliminating Virtual Function Calls in C++ Programs", *Proc. ECOOP '96*, Springer Verlag LNCS vol. 1098, pp. 142–166.
2. L. O. Andersen and C. K. Gomard, "Speedup Analysis in Partial Evaluation (Preliminary Results)", *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, June 1992, pp. 1–7. (Also available as Research Report YALEU/DCS/RR-909, Department of Computer Science, Yale University, New Haven, CT.)
3. J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad, "Fast, Effective Dynamic Compilation", *Proc. SIGPLAN '96 Conference on Programming Language Design and Implementation*, June 1996, pp. 149–159.
4. V. Bala, E. Duesterwald, and S. Banerjia, "Transparent Dynamic Optimization: The Design and Implementation of Dynamo", Technical Report *HPL-1999-78*, Hewlett-Packard Laboratories, Cambridge, Mass., June 1999.
5. R. L. Bernstein, "Producing Good Code for the Case Statement", *Software—Practice and Experience* vol. 15 no. 10, Oct. 1985, pp. 1021–1024.
6. B. Calder, P. Feller, and A. Eustace, "Value Profiling", *Proc. 30th International Symposium on Microarchitecture*, Dec. 1997, pp. 259–269.
7. B. Calder, P. Feller, and A. Eustace, "Value Profiling and Optimization", *Journal of Instruction-Level Parallelism* 1 (1999), 1–6.
8. C. Consel and F. Noël, "A General Approach for Run-time Specialization and its Application to C", *Proc.* 23rd *Annual ACM Symposium on Principles of Programming Languages*, Jan. 1996, pp. 145–156.
9. P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Apporoximation of Fixpoints", *Proc. Fourth ACM Symposium on Principles of Programming Languages*, 1977, pp. 238-252.
10. J. Davidson and S. Jinturkar, "Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses", *Proc. SIGPLAN 94 Symposium on Programming Language Design and Implementation*, June 1994, pp. 186-195.
11. J. W. Davidson and S. Jinturkar, "Aggressive Loop Unrolling in a Retargetable Optimizing Compiler", in *Proc. CC'96: Compiler Construction*, April 1996.
12. B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. eggers, "DyC: An Expressive Annotation-Directed Dynamic Compiler for C", Technical Report UW-CSE-97-03-03, Jan. 1998 (updated May 1999).
13. B. Grant, M. Philipose, M. Mock, C. Chambers, S.J. Eggers, "An Evaluation of Staged Run-time Optimizations in DyC", *Proc. SIGPLAN '99 Conference on Programming Language Design and Implementation*, May 1999, pp. 293–304.
14. U. Hölzle and O. Agesen, "Dynamic vs. Static Optimization Techniques for Object-Oriented Languages", *Theory and Practice of Object Systems* 1(3), 1996.
15. N. D. Jones, C. K. Gomard and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice Hall, 1993.
16. M. Mock, M. Berryman, C. Chambers, and S. J. Eggers, "Calpa: A Tool for Automating Dynamic Compilation", *Proc. 2nd. ACM Workshop on Feedback-Directed Optimization*, Nov. 1999. Available as `http://www-cse.ucsd.edu/users/calder/fdo/fdo2-mock.ps`.
17. S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufman, 1997.

18. R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere, "`alto` : A Link-Time Optimizer for the DEC Alpha", Technical Report 98-14, Dept. of Computer Science, The University of Arizona, December 1998.
19. M. N. Wegman and F. K. Zadeck, "Constant Propagation with Conditional Branches", *ACM Transactions on Programming Languages and Systems* vol. 13 no. 2, April 1991, pp. 181–210.

Appendix A Sources of Improvements

The sources of performance improvements for these benchmarks are discussed below. There is, however, one caveat. In our system, value-profile-based specialization is carried out after function inlining. Because of this, the code structure encountered during specialization, and the functions associated with the specialized code fragments, may not always correspond to those of the source program. Due to space constraints we only report most important sources for improvements.

compress : Expression profiling is used to unroll a loop and identify non-conflicting memory operations. This information allows memory access coalescing [10].

gcc : Most of the improvement comes from knowing that one of the values in the function *note_stores* has the value 34 over 80% of the time, and from knowing that 70% of the time the third argument to the function *simplify_binary_operation* is 34.

go : Roughly half of the improvement comes from specializing a value in the function *j2more* to 0, which causes several conditionals to be eliminated. Most of the rest of the speedup comes from specializing a value in the function *playnextto* to 0.

ijpeg : Expression profiling is used to unrol a loop and simplify the code in the unrolled loop.

li : Sequences of independent conditionals in functions *xleval* and *sweep* are transformed so that the common case is tested first. A `switch` statement in the function *livecar* is transformed so that the common case did not have to go through a jump table.

m88ksim : Expression profiling is used to determine that three pointers in the function *alignd* are unaliased in the common case, allowing the elimination of several load and store instructions in that function. The function *killtime* is specialized for an argument of 1.

perl : The function *memmove* is specialized for the single byte move. The (internal) function *OtsDivide64Unsigned*, which emulates integer divison (since the Alpha does not have an integer division instruction), is specialized for the divisor 16.

vortex : Most of the improvement comes from knowing that a value in the function *Mem_GetWord* takes on the value -1 nearly 100% of the time.