Data Specialization

Todd B. Knoblock and Erik Ruf
Microsoft Research
One Microsoft Way, Redmond, WA 98052 USA
{toddk, erikruf}@microsoft.com

Abstract

Given a repeated computation, part of whose input context remains invariant across all repetitions, program staging improves performance by separating the computation into two phases. An early phase executes only once, performing computations depending only on invariant inputs, while a late phase repeatedly performs the remainder of the work given the varying inputs and the results of the early computations.

Common staging techniques based on dynamic compilation statically construct an early phase that dynamically generates object code customized for a particular input context. In effect, the results of the invariant computations are encoded as the compiled code for the late phase.

This paper describes an alternative approach in which the results of early computations are encoded as a data structure, allowing both the early and late phases to be generated statically. By avoiding dynamic code manipulation, we give up some optimization opportunities in exchange for significantly lower dynamic space/time overhead and reduced implementation complexity.

1 Introduction

Program staging transformations capitalize on the fact that the inputs to a computation often become known in a particular order, or vary at differing frequencies. If we can establish that a particular portion of the input context remains invariant over multiple executions of the computation, then we can use this information to execute invariant subcomputations only once, and repeat only those subcomputations that depend on varying inputs. Doing so improves performance when the time savings from repeated execution of the optimized computation exceed the cost of performing the optimization dynamically.

In simple cases of staging, such as loop invariant code motion [ASU86], a compiler can automatically recognize the frequency with which the inputs (variable definitions) to a computation (loop body) are altered, determine which subcomputations depend only on invariant inputs, and hoist them to an appropriate location. In more complex situations, some degree of programmer assistance is required. Typically, the programmer statically partitions the input context into fixed and varying subparts, and invokes the staging transformation to obtain code for the *early* (invariant subcomputations) and *late* (varying subcomputations) phases. During program execution, the program invokes the early phase when the fixed inputs become known, and the late phase whenever the varying inputs change. It is the programmer's responsibility to enforce the invariant that fixed inputs will not change across invocations of the late phase; the early phase must be reinvoked whenever the fixed inputs are altered.

A number of staging techniques [MP89, KEH93, LL94, EP94, CN96, APC$^+$96] are based on dynamic compilation. These approaches range from runtime instantiation of manually generated machine-code templates to runtime execution of full-blown optimizers and code generators, with a variety of template- and compilation-based mechanisms lying between these two extremes. Despite their differences, these systems share a fundamental characteristic: they all express the output of the early phase as object code, an approach we will call *code specialization.* A typical code specialization architecture has the following type signature:

$$\textit{Fragment} \times \textit{Input-Partition} \to$$

$$\underbrace{(\textit{Fixed-Inputs} \to \overbrace{(\textit{Varying-Inputs} \to \textit{Result})}^{\text{dynamically generated code}})}_{\text{statically generated dynamic optimizer}}.$$

The programmer selects a program fragment to be optimized, and partitions its inputs into those which are to be held fixed across invocations of the optimized code, and those which may vary. Statically applying the compiler to a program fragment and an input partition yields object code for a runtime optimizer.[1] This optimizer is dynamically invoked on the fixed subset of the fragment's inputs, producing optimized object code. This optimized code is then (repeatedly) invoked on the remaining input, achieving the same effect as the original program fragment, but more quickly. Allowing the runtime optimizer to generate arbitrary object code can achieve a high degree of optimization; code specializers often eliminate branches, unroll loops, and produce improved instruction schedules in addition to folding operations involving fixed input values. Aggressive optimization

PLDI '96 5/96 PA, USA
© 1996 ACM 0-89791-795-2/96/0005...$3.50

[1] In some template-based systems, this step is performed manually by the programmer, who uses the input partition to construct a set of templates and code to instantiate them at runtime.

may require that the optimized code be executed many times to repay the cost of generating it.

In this paper, we describe a more limited approach, *data specialization*, in which the dynamic optimizer does not emit object code, but instead emits a *cache* of specialized data values. Along with the fragment's inputs, this cache is then (repeatedly) passed as input to statically-generated code that performs the remainder of the fragment's computation and returns the final value:

$$\textit{Fragment} \times \textit{Input-Partition} \to \underbrace{(\textit{Fixed-Inputs} \to \textit{Cache})}_{\text{statically generated cache loader}} \times \underbrace{(\textit{Cache} \times \textit{Varying-Inputs} \to \textit{Result})}_{\text{statically generated cache reader}}.$$

Statically applying the compiler yields two object codes: an optimizer, or *cache loader*, that computes the cache, and an execution engine, or *cache reader*, that uses the cache to compute the final output. Because the cache loader and reader are generated directly from the fragment and input partition without knowledge of the fixed data values, data specialization cannot, in general, achieve the same degree of optimization as code specialization. For example, it cannot eliminate branches or unroll loops, unless the same elimination/unrolling can be used for all possible values of the fixed inputs. Data specialization trades this generality for several other other desirable characteristics:

- Rapid payback: Cache loading is very inexpensive, and is typically amortized away after only two executions of the cache reader.
- Low space overhead: Caches are typically quite small (tens of bytes), allowing data specialization to be used in applications requiring very large numbers of specializations.
- Simple implementation: Data specialization can be implemented entirely via source-to-source program transformation, all of which can be performed statically given only the program and input partition. The transformation is thus completely portable, and the transformed code requires no additional runtime support.

The remainder of this paper describes the implementation and use of an instance of data specialization based on caching the values of certain invariant expressions. Other implementations are possible, as the signature above merely requires us to construct the late phase statically. We envision placing other kinds of useful information in the cache (c.f., Section 7.2).

We begin, in Section 2, by giving a small example. Section 3 describes basic algorithms for constructing cache loader and reader code from an arbitrary program fragment, while Section 4 treats some more advanced aspects of this transformation. In Section 5, we describe our prototype data specializer for a subset of C and show its performance and overhead on a family of graphics programs. We conclude with discussions of related and future work.

```
dotprod(x1, y1, z1, x2, y2, z2, scale)
{
  if (scale != 0)
    return (x1*x2 + y1*y2 + z1*z2) / scale;
  else
    return ERROR;
}
```

Figure 1: A dot product program.

cache loader:

```
dotprod_load(x1, y1, z1, x2, y2, z2, scale, cache)
{
  if (scale != 0)
    return ((cache->slot1 = x1*x2 + y1*y2) + z1*z2)
           / scale;
  else
    return ERROR;
}
```

cache reader:

```
dotprod_read(x1, y1, z1, x2, y2, z2, scale, cache)
{
  if (scale != 0)
    return (cache->slot1 + z1*z2) / scale;
  else
    return ERROR;
}
```

Figure 2: Cache loader and reader programs for $\{z1, z2\}$ varying.

2 Example

Consider the simple program fragment shown in Figure 1. If we expect to repeatedly execute this code while varying only the z coordinates, we may benefit by precomputing and caching the values of computations not depending on `z1` or `z2`, namely `(scale!=0)` and `(x1*x2+y1*y2)`.

For the sake of efficiency, our implementation constructs a loader and reader with signatures that differ slightly from those in the introduction in that (1) the loader and reader both receive all of the fragment's inputs as parameters, and (2) the loader returns both a cache and a result value:

$$\begin{aligned}&\textit{Fragment} \times \textit{Input-Partition} \to\\ &\quad (\textit{All-Inputs} \to \textit{Cache} \times \textit{Result}) \times\\ &\quad (\textit{Cache} \times \textit{All-Inputs} \to \textit{Result}).\end{aligned}$$

Case (1) allows the optimization of not caching information that can be cheaply recomputed from the fixed inputs, while (2) reduces overhead by allowing some computations in the loader to be used both for cache construction and result generation. Thus, the loader is essentially an instrumented version of the original fragment, while the reader is an optimized version. We also make use of heuristics; e.g., the loader does not cache `(scale!=0)`, as the relational operation is likely to be cheaper than a memory reference, but does cache the result of `(x1*x2+y1*y2)`; the reader then references the cached value instead of repeatedly computing `(x1*x2+y1*y2)`.

Invoking our data specializer on the fragment of Figure 1 produces the loader and reader code of Figure 2. These fragments illustrate two features of data specialization. First, because the loader and reader are constructed solely from the input partition by a process which does not have access to the value of `scale`, the conditional cannot be folded out, and appears in the reader. Second, the cache is small, containing only one value, and its initialization is very simple, adding only one assignment expression to the original program. A code specializer could eliminate the conditional, but would generate a larger specialization containing not only the value of `(x1*x2+y1*y2)`, but also opcodes for addition, multiplication, and division.

For this trivial program, we achieve a modest speedup (11% when `scale` is nonzero, 0% otherwise) by trading two multiplications and an addition for a memory reference. Had the basic operations been more expensive (e.g., matrix multiplications), the speedup would have been higher. Equally importantly, the startup cost is low (5.5% when `scale` is nonzero, 0% otherwise). Thus, we achieve breakeven whenever the original fragment is executed at least twice.

3 Specialization

In this section, we describe an algorithm for data specialization based on precomputing and caching the values of invariant expressions. This algorithm constructs the cache loader and cache reader from an imperative program fragment expressed as an abstract syntax tree and an input partition describing which variables are fixed on entry to the fragment. Specialization begins with analyses that annotate every term in the fragment with one of the following labels.

Static: the term only needs to be evaluated in the loader.

Cached: the loader must evaluate this term and load the resulting value into the cache. The reader does not evaluate this term, but instead reads the value from the cache.

Dynamic: both the loader and reader must evaluate this term (although its subterms may be recursively transformed).

The intuition behind these labels is that a term must be dynamic (i.e., appear in the reader) if its value depends in any way upon a "varying" input, or if its effects are visible to another dynamic term. A term is cached if it is not dynamic, but has a direct dynamic consumer. If neither of these applies, then a term is static, and is omitted from the reader. The cached terms represent the *maximal non-dynamic terms.* All of the dynamic terms are separated from the static terms by a frontier of cached terms. The loader and reader communicate only via these cached terms.

The conditions of this intuitive specification can be separated into two categories: (1) dependence upon the values of varying inputs and (2) consumption by dynamic terms. Our algorithm separates these conditions into two separate passes called *dependence analysis* and *caching analysis.* A third consideration, the efficient use of cache space, interacts with caching analysis, but will be deferred until Section 4.3 to avoid complicating the discussion. Once these analyses are complete, a simple *splitting transformation* is used to construct the cache loader and reader.

3.1 Dependence Analysis

Dependence analysis determines, for each program term, whether its result value (and, in the case of side-effecting terms, its effects), may depend upon the value of any of the varying inputs. Such *dependent* terms will have to be evaluated each time the reader is executed, while it may be possible to evaluate *independent* terms only once and cache their results. Dependence cannot be computed precisely; our approximation will (safely) err on the side of overestimating the set of dependent terms.

Given standard data dependence and control dependence information, computing our dependence relation is straightforward. A term is dependent if

1. it is a member of the varying part of the input partition,
2. it has a dependent operand,
3. it is reached by a dependent definition, or
4. it is conditionally reached by a definition along a path that is control dependent on a dependent predicate.

Cases (1) through (3) are straightforward: if a term directly or indirectly consumes the value of a varying input, it is dependent. Case (4) handles the situation in which a variable is conditionally set to one of multiple independent values, but the condition governing the choice cannot be evaluated given only the fixed input. This case is easy to recognize in a language having only structured control constructs because each join point corresponds to a single conditional, enabling us to force the appropriate variables (those modified in the single-entry, single-exit region associated with the join) to become dependent at the join point. In unstructured code, this problem becomes significantly more difficult; Auslander et al. [APC$^+$96] handle this case by performing a *reachability analysis* concurrently with dependence analysis.

Our implementation of dependence analysis is a straightforward, worst-case quadratic-time solution based on abstract interpretation. We initially assume that all terms other than the varying inputs are independent, then iteratively propagate the dependence constraints listed above. Other techniques for binding time analysis of imperative programs such as program slicing [DRH95] and type inference [And94] could also be applied here.

In the example program of Figure 1, the references to variables `z1` and `z2` are marked as dependent, as are the multiplication `z1*z2` and the surrounding addition and division. All other terms are marked as independent.

3.2 Caching Analysis

Caching analysis determines the structure of the cache loader and cache reader by annotating each program term as static, cached, or dynamic. This annotation is partially determined by the dependence analysis, which identifies terms whose value or effects may be influenced by the varying inputs. Since the execution of such dependent terms must be delayed until the values of the varying inputs are available, all dependent terms must appear in the cache reader; thus, the caching analysis annotates them as dynamic. The annotation is not completely determined by the dependence analysis due to two concerns:

- *Structural concerns:* terms in the reader may require the values or effects of other, possibly independent, terms, in order to execute. The caching analysis ensures that the requisite context will be available by caching the value or delaying the effects of such referenced terms. This concern is similar to the congruence criterion in offline partial evaluation [JGS93].

Any consistent cache labeling must satisfy the following system of constraints, where the functions *Dependent*, *Static*, *Cached*, and *Dynamic* are predicates on terms. Other predicates will be defined in the text accompanying each rule.

1. $\mathit{Dependent}(t) \rightarrow \mathit{Dynamic}(t)$
2. $\mathit{HasGlobalEffect}(t) \rightarrow \mathit{Dynamic}(t)$
3. $\mathit{UnderDependentControl}(t) \rightarrow \mathit{Dynamic}(t)$

Rules 1–3 are base cases that force certain terms to be labeled as dynamic; all three rules depend only on the program fragment and the dependence analysis. Terms whose value or effect depends on the varying portion of the input must be dynamic (Rule 1), as must those that read or write global state such as input/output or volatile storage (Rule 2). To avoid hoisting that could cause the loader to perform unnecessary computations, Rule 3 requires that any term whose execution is guarded by a dependent predicate be dynamic. Implementations willing to tolerate such speculation may choose to weaken this rule.

4. $\mathit{IsRef}(t) \wedge \mathit{Dynamic}(t) \rightarrow \forall t' \in \mathit{Defs}(t)\ \mathit{Dynamic}(t')$
5. $\mathit{Dynamic}(t) \rightarrow \forall t' \in \mathit{Guards}(t)\ \mathit{Dynamic}(t')$

Rules 4 and 5 handle cases where forcing a term t to appear in the reader (by labeling it as dynamic) requires that other terms defining the execution context of t also appear in the reader. If a variable reference appears in the reader, all definitions reaching the reference must also appear (Rule 4). Similarly, all control constructs guarding a dynamic term must also be dynamic (Rule 5).

6. $\mathit{Dynamic}(t) \rightarrow \forall t' \in \mathit{ValueOperands}(t)\ (\neg \mathit{Dynamic}(t') \wedge \mathit{SingleValued}(t') \wedge \neg \mathit{Trivial}(t') \rightarrow \mathit{Cached}(t'))$
7. $\mathit{Dynamic}(t) \rightarrow \forall t' \in \mathit{ValueOperands}(t)\ (\neg \mathit{Cached}(t') \rightarrow \mathit{Dynamic}(t'))$

Rules 6 and 7 construct the frontier of cached terms that defines the interface between the loader and reader. The basic idea is that, if a term appears in the reader, then it needs to obtain reader-time values for all of its value-producing operands, either by executing those operands or by retrieving their values from the cache. Operands executed solely for their effects can be ignored, as Rules 1 and 4 will add them as necessary. Rule 7 permits any operand to be annotated as dynamic, while Rule 6 permits caching any operand that meets three restrictions:

- It isn't already dynamic.
- It returns a single value during the execution of the fragment. This category includes all expressions not inside loops, and all expressions that are invariant in all enclosing loops. This restriction ensures that a single cache slot will correctly summarize the value of the operand.
- It is sufficiently nontrivial. For example, constants and expressions with very low execution costs are not cached.

8. $\neg(\mathit{Dynamic}(t) \vee \mathit{Cached}(t)) \rightarrow \mathit{Static}(t)$

Rule 8 ensures that all terms are labeled. Nodes that are neither dynamic nor cached become static, and need not appear in the reader.

Figure 3: Consistency constraints for caching analysis

- *Policy concerns:* not all independent terms need to be cached. For example, a term's value may not be used by the reader, or reevaluating the term may cost less than a memory reference to the cache. The caching analysis avoids these cases while ensuring that the structural concerns are satisfied.

Our caching analysis works by representing these concerns as a system of constraints limiting the possible labelings of the program terms. We then find a solution to these constraints under the criterion that the reader contain as few computations as possible. Figure 3 gives a system of consistency constraints for caching annotations. The idea is to find a basis set of terms that must appear in the reader, then ensure that all of the (transitive) data and control dependence predecessors of such terms will be represented in the reader (either by themselves or some cached value). Rules 1–3 identify the basis set: namely, dependent terms, terms with global side-effects, and terms whose execution in the reader would lead to speculation. The closure over dependences is implemented by Rules 4–7. Rule 4 ensures that any definitions reaching a dynamic term are dynamic, Rule 5 ensures that any control constructs guarding a dynamic term are dynamic, and Rules 6 and 7 ensure that the operands to any dynamic term are either cached or dynamic. By only caching terms with dynamic consumers, we satisfy the structural requirement that a fringe of cached terms separate all static and dynamic terms, and the policy requirement every data value in the cache have at least one consumer in the reader.

The distinction between the dependent and dynamic annotations is an important one because it allows us to force a computation into the reader while still making use of it in the loader. Consider the case of an independent assignment statement which reaches multiple variable uses, some of which are dependent. The assignment must appear in the reader so that the dependent use will access the proper value. If we were to implement this by making the assignment statement dependent, the definition of dependence would force all of its uses to become dependent, and we would be required to evaluate the independent variable uses (and any independent computations enclosing those uses) in the reader.

We find a solution to the constraints of Figure 3 by treating them as rewrite rules. We use Rule 8 to initially label all terms as static. Rules 1–3 rely only on dependence information and can be executed once. Rules 4–7 are executed on a demand-driven basis (whenever we label an expression as dynamic, we check to see which of these rules apply). We resolve conflicts between rules 6 and 7 by always attempting to apply rule 6 first; this gives us our preference for caching terms over executing them in the reader.

This algorithm requires time proportional to the size of the program, since each term can be labeled at most three times. If we consider the annotations as elements of an ordering *dynamic* > *cached* > *static*, then the algorithm is monotonic. This makes the algorithm restartable, in that we can at any time relabel any expression with label l with a new label $l' > l$, continue the execution of Rules 5–7, and obtain the same result as if the expression initially had label l'. We employ this observation in Section 4.3 where we use it in an algorithm that limits the amount of memory consumed by the cache.

In the example of Figure 1, the term (x1*x2+y1*y2) is marked as cached, with all of its subterms marked as static. Everything else is marked as dynamic ((scale!=0) is dynamic because it is trivial).

3.3 Splitting Transformation

Once the caching analysis is complete, we traverse the annotated fragment and emit the cache loader and the cache reader. The splitting transform proceeds via a simple case analysis based upon the caching annotation at each term.

Static: the recursively split term is added to the loader. Nothing is added to the reader for this term.

Cached: the recursively split term is added to the loader, along with an assignment to the corresponding cache slot. The reader receives a term that reads from the corresponding cache slot.

Dynamic: the recursively split term is added to both the loader and the reader.

If the cache frontier contains n terms, the size of the loader is that of the original fragment plus n assignments used to load the cache. The reader is smaller than the original fragment, as n terms of the fragment have been replaced by references to cache slots. In practice, the sum of the loader and reader sizes has been less than twice the size of the fragment.

In the example of Figure 1, the cached term (x1*x2+y1*y2) is wrapped with a cache assignment in the loader, and is replaced by a cache access in the reader. The static terms x1*x2 and y1*y2 appear only in the loader. The remaining terms are dynamic, and appear in both phases.

4 Advanced Specialization

In this section we describe a number of optimizations and refinements to the specialization algorithms described in Section 3. Most of the worked described in this section is aimed at minimizing the size of the cached data. This is important to the practical applicability of data specialization.

```
x = f(1)
if (p)
  x = g(2);

if (q)
  h(x);
z = x;
```

Figure 4: Example code where caching both references of variable x would be redundant. The predicates p and q are assumed to be *independent*, the expressions f(1) and g(2) are marked *independent and static*, h(x) is *dynamic*, and the two uses of x are marked *cached*.

cache loader:

```
x = f(1);
if (p)
  x = g(2);

if (q)
  h(cache->slot1=x);

z = (cache->slot2=x);
```

cache reader:

```
if (q)
  h(cache->slot1);

z = cache->slot2;
```

Figure 5: Unoptimized cache loader and cache reader fragments.

4.1 Using SSA to Improve Caching of Variables

The analysis described above determines which *expressions* should be cached. The splitting transformation then substitutes load and cache read instructions at the location of the expression in the original program. Consider the special case where the expression is a simple variable reference. Notice that if the *same* variable with the same reaching definitions occurs twice, and is marked as cached in both instances, the result would be to cache that value twice. In the example of Figure 4, naively applying the previous algorithm would result in a redundant cache slot being assigned to the second use of x as shown in Figure 5.

If slot1 is filled at all, it is filled with the same value as is slot2. To avoid this problem, we preprocess the program to produce unique definitions at the join points of the control flow graph. The result is analogous to static single assignment (SSA) form [CFR$^+$91]. In particular, we share the property with SSA that every variable reference except for those in the introduced assignments (our analog of the "phi" nodes of SSA) have exactly one reaching definition.

We produce this form via a source to source transform on the program before performing the specialization analyses. Starting at each control flow split, we analyze the branches for possible effects to variables. At the join point, we insert statements of the form v=v for each variable that may have been affected within the control term. We then disallow caching of individual variables except those introduced via *cache loader:*

```
x = f(1);
if (p)
  x = g(2);

x = (cache->slot1=x);

if (q)
  h(x);

z = x;
```

cache reader:

```
x = cache->slot1;
if (q)
  h(x);

z = x;
```

Figure 6: Improved cache loader and cache reader fragments.

the transformation (i.e., the phi nodes). The result for our example code fragment is shown in Figure 6.

In practice, this optimization typically has only minor effects. However, in a few programs, it has reduced the size of the cached data to as little as half the original size.

4.2 Associative rewriting

Consider the expression (x1*x2+y1*y2+z1*z2) where x1 and x2 are dependent. If the addition operator associates to the left, both additions will be dependent, while if it associates to the right, only the first one will be. Our implementation optionally reassociates expressions to maximize the size of independent terms, increasing the number of computations that can be performed in the loader.[2] This operation is similar to "binding time improvement" techniques used in offline partial evaluation, and to the rank-ordered reassociation used in code motion optimizations [ASU86, BC94].

4.3 Cache size limiting

Each program term annotated as cached represents an intermediate result value that will be computed by the loader, placed in the cache, and later used by the reader instead of executing the term. Thus, caching a term exchanges the time cost of executing the term for the space cost of storing its result value. The analysis described in Section 3.2 avoids caching terms that are inexpensive to execute, but treats space as an infinite resource; any nontrivial term that can usefully be cached will be. This is unrealistic; we must ensure that the caches of all simultaneously live specializations fit in physical memory, as paging in a cached value is almost certainly slower than recomputing it.

The goal of cache limiting is to minimize the amount of computation in the reader given a bound on the size of the cache. We approximate the cost of *not caching* each cached term, and relabel the lowest-cost cached term to dynamic, repeating this process until the cache size falls below the specified bound:

```
while (cache size > bound) do
   compute cost of not caching each cached term
   let victim = the minimum-cost cached term
     label victim as dynamic
   reestablish constraints 4–7 of Figure 3
```

Relabeling a term as dynamic requires that we reestablish constraints 4–7 of Figure 3 to ensure that the reader will contain the necessary execution context for the newly dynamic term. This may widen the cache frontier, increasing the amount of cache space required. Even though the total cache size does not necessarily decrease on each iteration of the loop, the loop will eventually terminate because each term is relabeled at most twice (from static to cached, or cached to dynamic).

The central task is choosing the "victim" term from the frontier of cached terms. We would like to keep cache elements that are expensive to compute, and make dynamic those with the least utility (perhaps weighted by size). Further complicating this decision is that the cost and utility of the cache elements are not independent, but depend upon what is already being cached versus computed dynamically. Our heuristic begins by statically approximating the execution cost of every program term (c.f., [WMGH94]), combining the following factors:

- a static cost value for the term's operator (for example, the cost of + is 1, the cost of / is 9),
- the sum of the costs of computing all subterms,
- for terms in loops, a multiplier (5),
- for terms guarded by conditionals, a divisor (2).

Given execution cost estimates for each term, we approximate the cost of not caching a given term on the cache frontier as the term's execution cost plus the transitive effect from rules 4–7 of Figure 3. These costs include those of definitions of variables referenced by the term, and of required guards that are not already dynamic (the marginal cost of computing an already dynamic guard is zero). After the minimum-cost term has been relabeled as dynamic, we efficiently reestablish the consistency constraints by restarting the constraint solver of Section 3.2, and check to see if the desired bound has been achieved.

Although this algorithm is decidedly approximate, it appears, in practice, to preserve the most important elements on the cache frontier. Sample results of this cache limiting algorithm are presented in Section 5.4.

5 Results

In this section, we present empirical results obtained with our data specializer. This system processes a subset of the C language without pointers or goto, and assumes that the fragment to be specialized is a single nonrecursive procedure. These restrictions simplify the computation of control and data dependence, and eliminate the need for an alias analysis, but otherwise do not impact the specialization algorithms. All measurements were conducted using the Microsoft Visual C++ compiler version 4.0 on an Intel Pentium/100 CPU with 64 megabytes of physical memory.

Our benchmarks are shading procedures, or *shaders*, belonging to the interactive graphics rendering system described in [GKR95]. A shader computes the color value for

[2] Of course, computer arithmetic does not obey the usual mathematical associativity rules. However, in many applications, this is not significant. In those where it is, this feature may be turned off.

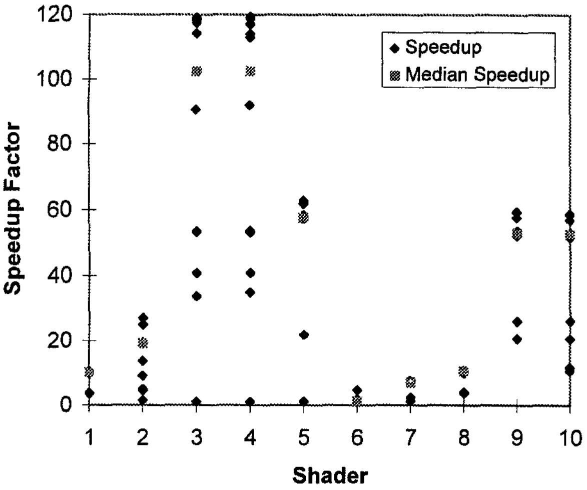

Figure 7: Speedup for all input partitions of ten sample shading procedures. Each shader is specialized on multiple input partitions (one per control parameter); these are displayed in the y direction above each shader number. Note that multiple input partitions having the same speedup are displayed as a single point.

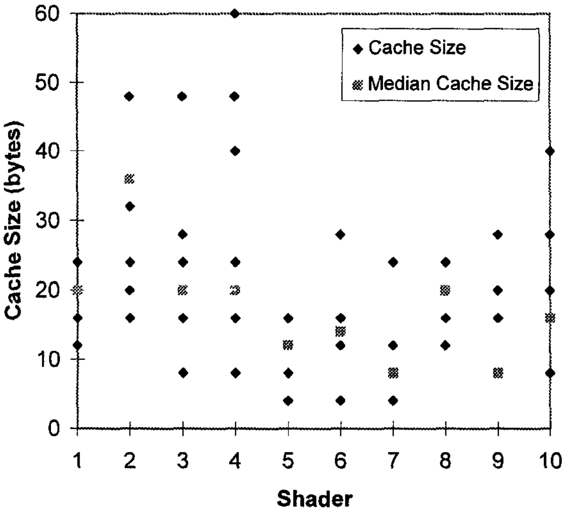

Figure 8: Single-pixel cache sizes for all input partitions of ten sample shading procedures. Note that multiple input partitions having the same cache size are displayed as a single point.

an image pixel given the pixel coordinates, various rendering information specific to the pixel, and shader-specific control parameters provided by the user via a graphical interface. The graphical interface restricts the user to modifying a single control parameter at a time, allowing us to specialize a shader on all of its inputs except for the control parameter being modified, and reuse the specialization (array of per-pixel caches) so long as the user continues to modify the same parameter. Because the fixed inputs include per-pixel rendering data, we may construct as many as 10^6 simultaneously live caches for a single image, but we require only one loader/reader code pair per input partition. A typical shader has on the order of 10 control parameters, requiring 10 loader/reader pairs. We construct, compile, and link this code statically at the time a shader is installed, an operation that takes only a few seconds per input partition.

In the sections that follow, we present results for ten shading procedures (some derived from examples in [Ups89, Smi90, GKR95], the remainder written by the authors) representing a variety of styles and complexity levels. These range in size from 50 to 150 lines of C code, and invoke a small mathematical library that supports vector and matrix operations as well as noise functions. We ported these shaders to our system as directly as possible without optimizing them for specialization.

5.1 Speedup

Figure 7 graphically depicts the asymptotic speedup achieved by specializing each of the 10 shaders with respect to input partitions holding all but one control parameter at a time fixed, yielding a total of 131 distinct input partitions. These values represent the average of multiple runs on a single image, with varying control parameter values. The speedups vary widely both between shaders and between input partitions of a single shader, but are alway at least 1.0x.

The high variance can be explained by the fact that the number and complexity of computations depending on the varying parameter (and thus the amount of work that must be performed by the reader) is different for each input partition. For example, shaders 3, 4, and 5 invoke expensive fractal noise functions; if the varying control parameter does not affect the input to the noise function, the noise value can be cached, and speedups as high as 100x are achieved. If, however, the noise function input is affected, the reader is forced to repeat this expensive computation every time the control parameter is altered, lowering the achievable speedup by approximately 50%. Simpler, non-iterative shaders such as 1, 6, 7, and 8 contain fewer expensive computations and thus exhibit lower speedups. However, their speedups still vary across input partitions: for example, changing the ambient light parameter (a simple scaling factor applied to the final color value) typically requires only a few multiplications, while altering the location of the light source affects virtually all of the shader's computations, including a number of coordinate transformations. Thus, a higher speedup is achieved for the ambient light parameter than for the light position parameters.

5.2 Overhead

The speedups described in the previous section are asymptotic, and do not reflect the additional cost of the extra cache loading operations that must be executed each time a new cache is required. For the user to benefit in practice, the loading cost must be amortized over multiple uses of the cache (i.e., several successive changes to a single shading parameter). Fortunately, this overhead is extremely low—of the 131 loader/reader pairs we constructed, 127 (97%) reached breakeven at two uses,[3] 3 required three uses, and 1 required 17 uses. It is also worth noting that our speedup and overhead figures are truly per-pixel statistics; we are not relying on a large image size to amortize costs.

[3] This means that the total time to shade a pixel twice using the loader/reader paradigm was no more than that required to shade that pixel twice using the original shading code.

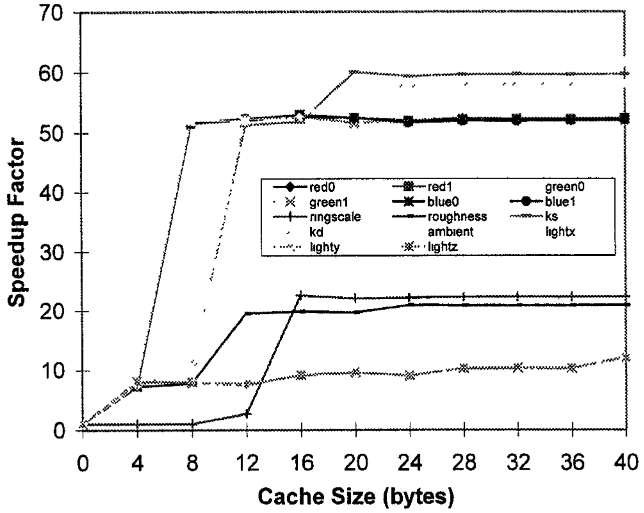

Figure 9: Speedup factor versus cache size for input partitions of shader 10.

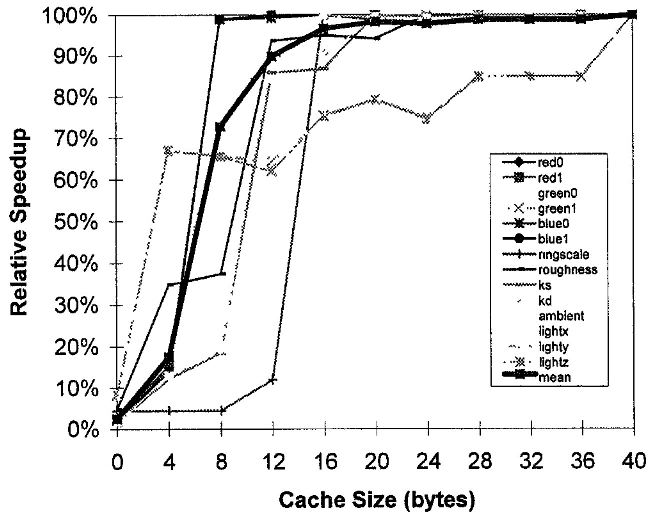

Figure 10: Percentage of maximum speedup achieved versus cache size for input partitions of shader 10.

5.3 Memory Usage

Figure 8 shows the number of bytes of cache space required for each pixel under each specialization of our sample shading procedures. As with the speedup figures, the cache size varies widely from specialization to specialization even for a single shading procedure. The overall mean and median cache sizes were 22 and 20 bytes, respectively. In every case, multiplying the cache size by the number of caches constructed (307,200 caches for a 640-by-480 image), yields a total space usage well within the physical memory size of a typical workstation.

5.4 Reducing Memory Usage

The cache limiting algorithm of Section 4.3 allows us to trade decreased cache size for decreased speedup by forcing more computations into the reader. Figure 9 demonstrates the absolute speedup factors achieved for various cache size limits applied to all 14 input partitions of shader 10. Figure 10 displays the same information where each input partition's maximum speedup is normalized to 100%. As the cache size limit is reduced from 40 bytes to 0 bytes, the speedups are reduced,[4] but a large fraction of the speedup is achieved even when the cache is greatly reduced in size. Overall, 70% of the performance is retained when the cache is limited to 20% of the maximum, while 90% is achieved when the limit is raised to 30%.

Two effects contribute to this result. First, many of the input partitions require fewer than 40 bytes of cache space, so they are not affected until the limit is moved to below their "natural" cache size. Second, some cacheable computations are more expensive than others; often, once the most critical computations have been cached, additional caching yields only a minor improvement. For example, the first 4-byte floating-point value cached by the specialization for the input partition with parameter **lightx** varying accounts for 65% of that specialization's speedup, even though maximum speedup requires 40 bytes. Our heuristic attempts to capture this second effect. This gradual degradation cannot always be achieved; for example, in the specialization in which **ringscale** is varying, reducing the cache size limit from 16 bytes to 12 bytes leads to a 95% reduction in speedup independent of which values are cached.

6 Related Work

Data specialization can be viewed as a staging transformation [JS86] that moves computations to contexts where they will be executed less often. In this section, we describe two other staging strategies: runtime code generation and incremental program execution. We also relate several of our implementation techniques to their counterparts in partial evaluation.

6.1 Runtime Code Generation

Several approaches to dynamic optimization rely on the fast instantiation of precomputed object code templates. Massalin and Pu [MP89] used hand-generated assembly templates; Consel and Noël [CN96] automatically generate portable templates at the source code level and extract them from the resulting compiled code using machine-specific techniques. Auslander et al. [APC$^+$96] present a template-based compiler for an annotated version of C; this system uses analyses similar to ours to construct an early phase that fills in a "run-time constants table" similar to our cache. Our systems differ in that we construct a single reader that references the cache, while Auslander et al. use the cached data to instantiate immediate values in dynamically generated code.

Non-template systems optimize and generate code from an intermediate form at runtime. Tools for this purpose include DCG [EP94] and 'C [EHK96], which have achieved speedups as high as 50x, but require tens to hundreds of dynamic instructions to emit a single optimized instruction. Leone and Lee [LL94] used partial evaluation to "compile out" the intermediate form, generating a custom optimizer

[4] The few small increases in speedup as cache size is decreased have been verified to be due to timing imprecision and processor cache effects rather than poor choices made by the heuristic, as the generated code was identical. The error appears artificially large for parameters lightx, lighty, and lightz in Figure 10 because their speedup range is smaller than those of other parameters.

that directly emits object code. Keppel et al. [KEH93] compared assembly-level and compiler intermediate representation (IR) level template compilers for a variety of workloads, and computed conservative amortization intervals of 10-1000 uses for assembly templates and 1000-infinite uses for IR templates.

General partial evaluators [JGS93, Ruf93, And94, Osg93] can yield highly specialized code, but existing systems are impractical for runtime use due both to their slowness and to the cost of dynamically compiling the high-level code they generate. Other dynamic compilation systems [DS84, Cha92] concentrate more on optimizing language features such as dynamic dispatch than on staging user-level computations.

An alternate formulation of partial evaluation known as mixed computation [Ers77, Bul84] has a signature somewhat similar to that of data specialization, in that the transformation emits both specialized code and specialized data. The difference is that our approach emits code statically and data dynamically, while systems based on mixed computation emit both simultaneously, requiring them to pay the cost of dynamic code generation if used at runtime.

6.2 Incremental Program Execution

Incremental program execution techniques effectively stage programs by caching intermediate results for re-use in subsequent executions. Systems that cope with arbitrary input changes by dynamically checking dependences [PT89, Hoo92] avoid more computations than data specialization does, but they lose the efficiency we gain from "compiling away" the dependences in advance.

Liu and Teitelbaum [LT95a, LT95b] present algorithms for statically deriving an incremental version of a pure functional program under some input change. The basic idea is to identify computations in a program whose values can be profitably reused when the program is reexecuted under the input change, and to cache these values instead of recomputing them. By using a programmatic description of the input change and taking advantage of algebraic identities, this method can support forms of reuse which our purely dependence-based algorithm cannot. For example, using the cached value of the expression $\mathit{fibonacci}(x-1)$ in place of the expression $\mathit{fibonacci}(x-2)$ under the input change $(\lambda x.x+1)$ yields a form of finite differencing.

Sundaresh and Hudak [SH91] derive incremental versions of functional programs by expressing the program as the composition of a number of residual program fragments, each of which depends on a different projection of the program's inputs. When an input is changed, the system rebuilds the corresponding fragment, merges the code for all fragments, and executes the result. This is a form of code specialization as it requires the dynamic construction (and compilation) of code whenever an input is changed.

6.3 Partial Evaluation

Several aspects of the implementation of data specialization described in Section 3 are similar to techniques used in partial evaluation. Our dependence annotation is similar to the binding time attribute computed by offline partial evaluators [JGS93], in that both involve transitive data dependence on distinguished inputs. The approaches differ in that we separate semantic (dependence) information from policy (caching) information, while binding time analyzers typically mix both in the binding time attribute. We have found that the latter approach can introduce false dependences. For example, our caching analysis can label a term as dynamic without forcing its consumers to be dynamic, while a BTA-based approach (in which $\mathit{dependent} \equiv \mathit{dynamic}$) would unnecessarily force all of the term's consumers into the reader. We believe that current partial evaluators for imperative languages [Osg93, And94] have not yet experienced this problem because they do not perform flow-sensitive binding time analysis.

Program bifurcation [Mog89, DNBDV91] factors each of a program's functions into two new functions: (1) a function which takes as input only independent values, and produces the independent portion of the result, and (2) a function which takes both independent and dependent inputs and produces only the dependent portion of the result. An effect similar to data specialization could be obtained by caching certain intermediate results of type (1) functions and using the corresponding values in the type (2) functions.

Our splitting pass, which traverses the annotated program fragment and emits the loader and reader code, can be viewed as two nonstandard semantic interpretations of an action tree [CD90, CN96]. Consel's "evaluate" action denotes maximal-sized independent subtrees and thus corresponds (in the absence of speculation avoidance and cache size limiting) to our cached annotation. Because we require that a single reader suffice for all potential values of the nonvarying inputs, all three of Consel's "rebuild," "reduce," and "identity" annotations correspond to the dynamic annotation in our system.

7 Future Work

We foresee a number of ways to extend this work, both in terms of the present implementation and the broader framework. We are also actively seeking other applications that will benefit from our approach.

7.1 Extending the Implementation

Expressing our transformation in terms of expressions (abstract syntax trees) is convenient for expository purposes but difficult to implement, particularly in the face of side effects and nonlocal control transfers. We expect to move to a control flow graph representation in the near future.

We would like to explore the costs/benefits of allowing speculation in the loader. Because the load-time overhead is presently very low, we can probably afford the time overhead of extra, potentially-unused computations in the loader; other potential problems include the additional cache space required to store the result values and the extra work required to trap and handle exceptions.

In our current architecture, we perform staging offline as a source-to-source transformation; this limits the number of distinct input partitions we can handle. By computing the necessary control and data dependence information offline (e.g., manually staging our staging transformation!), we may be able to perform our analyses and transformations (including code generation) dynamically.

7.2 Extending the framework

Reifying the result of the early phase of a staged program as a data structure need not limit us to caching intermediate results. For example, we might choose to combine the result of several control transfers into a single index into a lookup table, and cache only the index value. We could also speculatively construct multiple specialized cache readers targeted to particular fixed input values and select among them using a dispatch code passed in the cache.

We might also find it fruitful to explore points in the spectrum between data specialization (which emits no code at runtime) and code specialization (which expresses all early results as runtime-generated code). This would allow us the optimization benefits of code specialization for long-lived, often-reused contexts, while retaining the low space/time overhead of data specialization for more ephemeral contexts.

7.3 New Applications

To date, we have only experimented with programs from a single domain, namely local (per-pixel) shading computations in graphics rendering, which have the characteristics:

1. a repeated computation has an input context whose components vary at different rates,
2. a sufficient fraction of the computation depends only on a subset of the inputs,
3. the invariant portion of the computation can be usefully encoded in the form of cached intermediate result values, and
4. the space/time overhead requirements of the application preclude the use of the more general code specialization staging technique.

We expect that other applications also meet these criteria, and will benefit from our technique. In particular, item (3) suggests that we concentrate on numeric applications where significant effort goes into the production of a small number of values, rather than on interpreter-like applications where the early computation primarily performs dispatch rather than computing values. Item (4) suggests applications that either require a large number of simultaneous specializations, such as image processing, or those where the repetition count is likely to be low, such as those where the fixed parameters are derived from interactive user input.

8 Conclusion

We have presented a new technique, data specialization, for improving the performance of program fragments by automatically restaging them into an optimizer and an execution engine that communicate via a cache of data values. This approach complements existing techniques based on runtime code generation; although it achieves a somewhat lower degree of optimization, its very low overhead in time and space make it attractive in applications where existing techniques cannot be profitably applied.

Acknowledgements

Brian Guenter first suggested that we apply this technique to shading, and collaborated on the implementation of the interactive shading system described in Section 5. Bruce Duba, Linda O'Gara, Thomas Reps, Daniel Weise, and the anonymous referees provided helpful comments on drafts of this paper.

References

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, Denmark, 1994. DIKU Research Report 94/19.

[APC$^+$96] Joel Auslander, Matthai Phillipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation.* ACM, May 1996. This volume.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers; principles, techniques, and tools.* Addison-Wesley, Reading, MA, 1986.

[BC94] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 159–170, June 1994.

[Bul84] M.A. Bulyonkov Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[CD90] Charles Consel and Olivier Danvy. From interpreting to compiling binding times. In N. Jones, editor, *Proceedings of the 3rd European Symposium on Programming*, pages 88–105. Springer-Verlag, LNCS 432, 1990

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[Cha92] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Stanford University, March 1992. Published as technical report STAN-CS-92-1420.

[CN96] Charles Consel and Francois Noël. A general approach to run-time specialization and its application to C. In *Proceedings 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156. ACM, January 1996.

[DNBDV91] Anne De Niel, Eddy Bevers, and Karel De Vlaminck. Program bifurcation for a polymorphically typed functional language. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 142–153. New York: ACM, 1991.

[DRH95] Manuvir Das, Thomas Reps, and Pascal Van Hentenryck. Semantic foundations of binding time analysis for imperative programs. In *Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, June 1995.* New York: ACM, 1995.

[DS84] L. Peter Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302. ACM, 1984.

[EHK96] Dawson Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C, a language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 131–144. ACM, January 1996.

[EP94] Dawson R. Engler and Todd A Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 263–272, October 1994.

[Ers77] Andrei P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, April 1977.

[GKR95] Brian Guenter, Todd B. Knoblock, and Erik Ruf. Specializing shaders. In *ACM SIGGRAPH'95 (Computer Graphics Proceedings, Annual Conference Series)*, pages 343–349, 1995.

[Hoo92] Roger Hoover. Alphonse: incremental computation as a programming abstraction. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 261–272, San Francisco, CA, June 1992.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[JS86] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida*, pages 86–96. New York: ACM, 1986.

[KEH93] David Keppel, Susan J. Eggers, and Robert R. Henry. Evaluating runtime-compiled value-specific optimizations. Technical Report UWCSE 93-11-02, University of Washington Department of Computer Science and Engineering, 1993.

[LL94] Mark Leone and Peter Lee. Lightweight runtime code generation. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 97–106, 1994.

[LT95a] Yanhong A. Liu and Tim Teitelbaum. Caching intermediate results for program improvement. In *Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, June 1995*, pages 190–201. ACM, 1995.

[LT95b] Yanhong A. Liu and Tim Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24:1–39, 1995.

[Mog89] Torben Mogensen. Separating binding times in language specifications. In *Fourth International Conference on Functional Programming Languages and Computer Architecture, London, England, September 1989*, pages 14–25. Reading, MA: Addison-Wesley, 1989.

[MP89] Henry Massalin and Calton Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[Osg93] Nathaniel David Osgood. PARTICLE: an automatic program specialization system for imperative and low-level languages. Master's thesis, MIT, September 1993.

[PT89] William Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 315–328, Austin, TX, January 1989.

[Ruf93] Erik Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, California, April 1993. Published as technical report CSL-TR-93-563.

[SH91] R.S. Sundaresh and Paul Hudak. A theory of incremental computation and its application. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–13, January 1991.

[Smi90] Alvy Ray Smith. Unpublished notes. 1990.

[Ups89] Steve Upstill. *The RenderMan Companion*. Addison-Wesley, 1989.

[WMGH94] Tim A. Wagner, Vance Maverick, Susan Graham, and Michael A. Harrison. Accurate static estimators for program optimization. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 85–96, June 1994.