Resource-Bounded Partial Evaluation *

Saumya Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721, U.S.A.
debray@cs.arizona.edu

Abstract

Most partial evaluators do not take the availability of machine-level resources, such as registers or cache, into consideration when making their specialization decisions. The resulting resource contention can lead to severe performance degradation—causing, in extreme cases, the specialized code to run slower than the unspecialized code. In this paper we consider how resource considerations can be incorporated within a partial evaluator. We develop an abstract formulation of the problem, show that optimal resource-bounded partial evaluation is NP-complete, and discuss simple heuristics that can be used to address the problem in practice.

1 Introduction

The field of partial evaluation has matured greatly in recent years, and partial evaluators have been implemented for a wide variety of programming languages [1, 4, 5, 6, 20, 29]. A central concern guiding these implementations has been to ensure that input programs should be specialized as far as possible without compromising termination of the partial evaluator. The good news is that most current implementations of partial evaluators have achieved some success at satisfying this concern. Unfortunately, the bad news is also that these systems are successful at meeting this concern. Focusing single-mindedly on specializing as much of an input program as possible, partial evaluators typically ignore the availability of machine-level resources, such as registers or cache, when making specialization decisions. The resulting resource contention can lead to significant performance degradations—in some cases, causing the specialized program to run slower than the unspecialized code.

The problem is illustrated by Figure 1. This figure illustrates how the speedup of a convolution-like program, which computes $\sum_{j=1}^{n}\sum_{i=1}^{n} x_i y_j$ given two n-element integer vectors x and y and which has been specialized to one of the input vectors, varies for different values of n.[1] It can be seen that while the specialized program is about 25% faster than the unspecialized version for small values of n, the speedups drop off steeply after $n = 4000$, and for $n \geq 7000$ the specialized code is slower than the unspecialized program.

* A preliminary version of this paper appeared in *Proc. 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)*. This work was supported in part by the National Science Foundation under grant CCR-9502826.

[1] The numbers are based on a Scheme program that represents a vector as a list, specialized using Similix [6] and compiled using Bigloo version 1.8 [30], with gcc version 2.7.2 as the back-end compiler, and run on a 25 MHz SPARC IPC with 64 Kbytes of cache and 32 Mbytes of main memory. See Section 5 for further details.

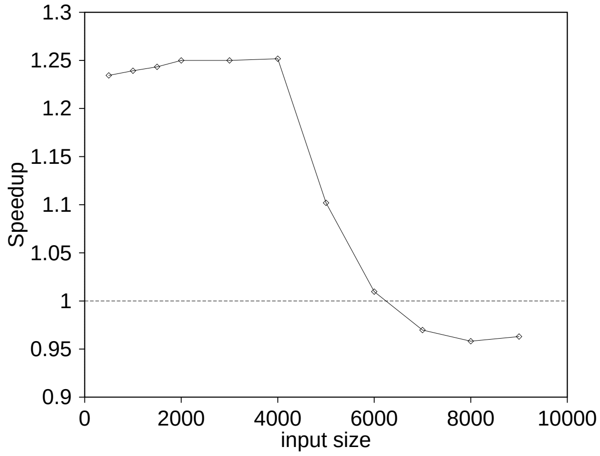


Figure 1: Relative Performance of Specialized Convolution Programs

This loss in performance is due not so much to an "explosion" in code size as it is to specialization with no regard for machine-level resources. What happens is that because one of the input vectors is known at specialization time, the inner loop of the program is unrolled completely into a straight line sequence of code. For different input sizes, this results in a family of specialized programs where the size of the body of the main loop—where most of the execution time of the program is spent—grows linearly as n. This apparently modest growth rate incurs significant performance penalties once n becomes large enough that the body of the loop does not fit in the instruction cache of the processor.

There are two reasons why performance degradation due to over-specialization can be a problem. First, as the example given illustrates, it can manifest itself even for commonly encountered computations and reasonably sized inputs; and second, the performance degradation typically increases as the (static) input size increases and is the worst for the largest inputs, whereas these are precisely the cases where we would like partial evaluation technology to deliver the greatest benefits. For these reasons, it would seem desirable to be able to incorporate some awareness of the availability of resources during the partial evaluation process.

The potential problem of code growth during partial evaluation has been noted in the past (e.g., see [24]), but we do not know of any proposal to address code growth given fixed resources. There has been some work on identifying those static computations whose specialization can contribute to good speedups [2], but these do not directly address the particular problem that we are concerned with. The issue of resource-bounded partial evaluation has been discussed by Danvy *et al.* [11], who sketch possible approaches to the problem at a very high level but offer few details. Many of the issues that arise in this work have been considered in the compiler optimization literature as well, albeit in considerably more restricted contexts. This includes work on function inlining [7, 12, 14, 15, 23], loop unrolling [9, 13, 16, 31], and improving cache utilization of programs [22, 25, 28].

2 Underlying Concepts

We assume that the programs being specialized are expressed in a (untyped, eager) first-order functional language. We will generally work with binding-time annotated programs with the following abstract syntax:

$P \in \mathsf{Prog}$	(programs)
$e \in \mathsf{Expr}$	(expressions)
$x \in \mathsf{Var}$	(variables)

$$P ::= \{f_1(x_1, \ldots, x_n) = e_1, \ldots, f_n(x_1, \ldots, x_n) = e_n\}$$

$$
\begin{array}{lll}
e ::= & c & \text{(constant)} \\
 & \mid x_s & \text{(static variable)} \\
 & \mid x_d & \text{(dynamic variable)} \\
 & \mid op_s(e_1, \ldots, e_n) & \text{(static base operation)} \\
 & \mid op_d(e_1, \ldots, e_n) & \text{(dynamic base operation)} \\
 & \mid \mathtt{if}_s(e_1,\ e_2,\ e_3) & \text{(static conditional)} \\
 & \mid \mathtt{if}_d(e_1,\ e_2,\ e_3) & \text{(dynamic conditional)} \\
 & \mid \mathtt{lift}(e) & \text{(lift a static expression)} \\
 & \mid \mathtt{let}_s\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 & \text{(static let expression)} \\
 & \mid \mathtt{let}_d\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 & \text{(dynamic let expression)} \\
 & \mid \mathtt{call}_s(f, e_1, \ldots, e_n) & \text{(static call)} \\
 & \mid \mathtt{call}_d(f, e_1, \ldots, e_n) & \text{(dynamic call)}
\end{array}
$$

The body of a function f will sometimes be denoted by $body(f)$. It is assumed that all, and only, static function calls are considered for unfolding. For concreteness, we will informally use a first-order subset of Scheme for our examples.

2.1 Program Points

Let a *control point* refer to any executable construct within a program, and a *static environment* at a control point refer to a mapping from the static variables at that point to values. Traditionally, a "program point" in a specialized program is taken to be a pair $(cp, senv)$, where cp is a control point in the original (i.e., unspecialized) program and $senv$ a static environment for that point. For our purposes, we need to extend this so that a program point in a specialized program is a pair $(cp, SEnv)$ where $SEnv$ is a *set* of static environments corresponding to the control point cp in the original program. To see the reason for this, consider the following Scheme code:

```
(define (foo y)
  (define (f y i)
    (if (= i 0) y
        (let* ( (y0 (+ y i)) (i0 (- i 1)) )
              (f y0 i0)))
  )
  (f y 10000)
)
```

The sort of code we would intuitively like to generate is that obtained by unfolding the recursive call to `f`, as much as possible while ensuring that the body of the resulting code still fits in the cache, i.e., something like

```
(define (foo y)
  (define (f y i)
    (if (= i 0) y
        (let* ((y0_0 (+ y i))  (i0_0 (- i 1))
               (y0_1 (+ y0_0 i0_0))  (i0_1 (- i0_0 1))
               (y0_2 (+ y0_1 i0_1))  (i0_2 (- i0_1 1))
                  ...
               (y0_99 (+ y0_98 i0_98))  (i0_99 (- i0_98 1))
              )
            (f y0_99 i0_99)))
```

```
    )
    (f y 10000)
  )
```

However, if we take a specialized program point to be a (control point, static environment) pair, then specialization will proceed with the sequence of program points $(L, [\texttt{i} \mapsto 10000]), (L, [\texttt{i} \mapsto 9999]), \ldots$, where L denotes the control point where `y0` is bound to `(+ y i)`, and this will result in the generation of the specialized statements with the static values of the variable `i` hard-wired in, as follows, which is not what we want:

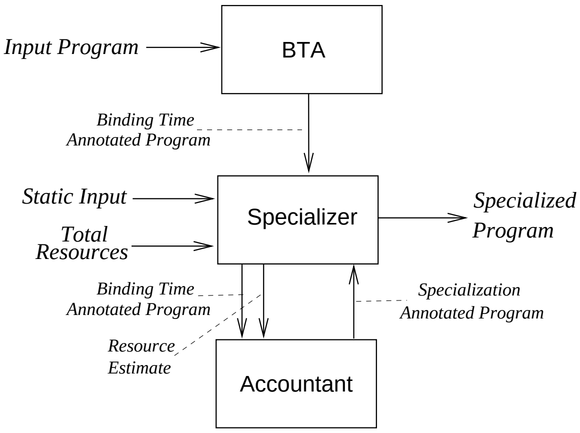
```
(define (foo y)
   (define (f y)
      (let* ( (y0_0 (+ y 10000))
              (y0_1 (+ y0_0 9999))
              (y0_2 (+ y0_1 9998))
                ...
            )
            y0_10000
      )
   )
   (f y)
)
```

This problem can be avoided by having a program point in a specialized program associate a set of static environments with each control point in the original program: in the special case where this set is a singleton, the values of the static variables can be substituted for the variables, as before.

If the available resources do not allow a program to be specialized to the extent where there is exactly one static environment associated with each specialized program point, then those points that are associated with more than one static environment cannot have the values of the corresponding static data hard-wired into the residual code. For such points, it is necessary to retain the values of the static data separately, and we need to take into account the resources necessary for this when estimating the amount of resources actually available for accommodating code growth during partial evaluation (if instructions and data compete for resources, it makes sense to try and have both the code and the data fit in the available resources if possible: a cache miss due to a data reference is just as expensive as a miss due to an instruction reference). On the other hand, if the program can be specialized to the extent that there is at most one static environment per specialized program point, the static data values can be hard-wired into the residual program, making it unnecessary to store them separately, and potentially freeing up additional space for code growth. In other words, if we cannot be sure that the program can be specialized "all the way" and still fit within the available resources, we should allow for the fact that some of the static data will have to be kept around, and that as a result, not all of the resources may be available to accommodate code growth. This implies also that if we are able to detect situations when the static data need not be kept around separately, it may be possible to do a better job of specialization than if we conservatively retain space for the static data.

2.2 Specialization Annotations

Traditional offline partial evaluation consists of two components: a *binding time analyzer*, which determines which computations can be specialized; and a *specializer*, which carries out the actual specialization, based on the information provided by the binding time analyzer. To account for resource availability, we extend this to include a third component, the *accountant*, as depicted in Figure 2. The idea is that the binding time analyzer determines which computations can be specialized and passes this information—in the form of a program annotated with binding times—to the specializer. The specializer then queries the accountant



(a) Organization

Input : An input program P_{in}, resource bound *ResourceAvail*.

Output : A specialized program P_{out}.

Method :

$P_0 := BindingTimeAnalysis(P_{in})$;
repeat
    $P_1 := Accountant(P_0, ResourceAvail)$;
    $S :=$ set of specialization annotated points in P_1;
    if $S \neq \emptyset$ **then**
        $(P_2, ResourceUsed) := Specialize(P_1)$;
        $ResourceAvail := ResourceAvail - ResourceUsed$;
        recompute static environments for functions where unfolding has occurred;
        $P_0 := P_2$;
    fi
until $S = \emptyset$;
$P_{out} :=$ remove binding time annotations from P_0;
return P_{out};

(b) Algorithm

Figure 2: Overview of Resource-Bounded Partial Evaluation

with a binding-time annotated program, which indicates which operations *can* be specialized, together with an estimate of the available resources. The accountant uses the resource information to determine which operations *should* be specialized, and tags each static computation with one of the annotations *specialize* or *don't-specialize* (another way to think of this is to imagine the accountant as turning some of the "static" annotations to "dynamic" to prevent some specializations from taking place). This specialization-annotated program is returned to the specializer, which carries out a single "specialization step" (see below) based on the decisions of the accountant and updates its resource availability estimates accordingly. Any new control points in the resulting program are annotated with binding time annotations inherited from the original binding-time annotated program where necessary, and static environments associated with such points is determined. This is then used to query the accountant again, together with the updated resource estimates. This process continues, with the specializer repeatedly querying the accountant with a binding-time annotated (partially specialized) program together with an estimate of the available resources, then specializing the specialization-annotated program returned by the accountant and updating its resource estimates, until the accountant terminates the specialization process due to the exhaustion of available resources (or possibly because it identifies a nonterminating specialization sequence) by returning a program where no operation is annotated for specialization.

An important notion in this context is that of a *specialization step*, which refers to the actions the specializer can take unilaterally between two consecutive queries to the accountant. Since we are concerned primarily with code growth, which occurs when function calls are unfolded, we have to limit the number of unfolding steps that can occur between any two consecutive queries to the accountant. We therefore define a specialization step to be a (maximal) sequence of reductions such that, whenever a function call is unfolded, calls occurring in the unfolded body are not considered for further unfolding.

In some ways, the use of specialization annotations resembles the notion of "mixline" partial evaluation [10], where binding-time annotations of "possibly static" or "sometimes static" are permitted. The difference is that in our model, the binding time analysis does not itself distinguish between "possibly static" and "definitely static" entities: it identifies everything that is (definitely) static, and the accountant selects a subset of these for actual specialization in any particular specialization step based on the availability of resources. In the degenerate case where the accountant ignores the availability of resources and always selects all static computations for specialization, this model becomes indistinguishable from the traditional approach to partial evaluation.

Traditionally, the classification of variables as "static" or "dynamic" is required to satisfy a congruence condition that states, essentially, that any variable that depends on a dynamic variable is itself dynamic. The reason for this is that if the value of a variable is to be computable before the program is executed, that value cannot be dependent on any quantity that is not available until runtime. It is not difficult to see that the specialization-annotated program must satisfy a similar condition, since any operation that is being specialized must have its operands available, which means that anything it depends on must also be specialized.

2.3 An Abstract Formulation of the Problem

Intuitively, a resource-bounded partial evaluator will attempt to specialize as much of an input program that it can, subject to the usual termination considerations as well as considerations of the availability of resources. For this, it will need to be able to weigh the benefits of specializing a particular computation against the costs so incurred. When a piece of code is specialized with respect to some static data, it will typically be the case that the residual code will require fewer operations, measured, for example, in instructions or function calls. The savings so incurred must be weighed against the storage requirements of the residual program, measured, for example, in the number of registers required for live values, or in the number of instructions that need to reside in cache. Specialization of an expression may lead to a reduction in code size (if some operations are specialized away) or an increase in code size (if specialization involves unfolding a function call; or leads to a primitive operation being open-coded instead of being implemented as a call to a generic routine). Thus, with each control point p we can associate a cost $\mathsf{cost}(p) \in \mathcal{Z}$ and a benefit $\mathsf{savings}(p) \in \mathcal{Z}$, where $\mathcal{Z}$ denotes the set of integers. Moreover, given limited resources we will prefer to focus on those parts of the program that are the most frequently executed: to this end, we assume that each point p has a nonnegative "weight" $\mathsf{wt}(p)$ associated with it.

In general, the problem of resource-bounded partial evaluation for a resource bound of B would involve coming up with a specialization sequence, i.e., sequence of specialization steps, for any given program such that (i) the total savings is maximized, and (ii) the size of the residual program does not exceed B. This does not seem to be a straightforward problem: for example, the program resulting from an intermediate specialization step can be allowed to exceed the bound B—perhaps by a considerable amount—as long as enough code can specialized away subsequently to reduce the size of the final residual program to below B. This would appear to involve a search for a global optimum over the space of all possible specialization sequences, and it is not obvious that this will be practical, especially for nontrivial programs. We therefore consider a stronger criterion, namely, that each specialization step in a specialization sequence should respect the resource bound B. We refer to such specialization sequences as *pointwise resource-bounded*; An optimal pointwise resource-bounded specialization sequence is one that is pointwise resource-bounded, and where at each step the savings are maximized.

3 Complexity Issues

It is easy to see that one way to obtain an efficient algorithm for resource-bounded partial evaluation is to focus on pointwise resource-bounded specialization sequences, using an efficient algorithm for each specialization step, and ensuring that the specialization sequences are not too long (i.e., are within a polynomial factor of the size of the input program). In this section we focus on the complexity of a single optimal resource-bounded specialization step. This optimization problem can be rephrased as a decision problem as follows:

Definition 3.1 The Optimal One-Step Resource-Bounded Specialization problem is defined as follows: given a set of control points P; functions $\mathsf{wt} : P \to \mathcal{Z}$, $\mathsf{cost} : P \to \mathcal{Z}$ and $\mathsf{savings} : P \to \mathcal{Z}$; and positive integers B, K, is there a set of points $Q \subseteq P$ satisfying the following requirements:

(i) if $q \in Q$ and q depends on p then $p \in Q$;
(ii) $\sum_{q \in Q} \mathsf{wt}(q) \cdot \mathsf{savings}(q) \geq K$; and
(iii) $\sum_{q \in Q} \mathsf{cost}(v) \leq B$?

∎

The structure of this problem, where we try to maximize one quantity while simultaneously trying to minimize another, is reminiscent of "knapsack"-like problems. The following result therefore does not come as a great surprise:

Theorem 3.1 *Optimal One-Step Resource-Bounded Specialization is NP-complete in the strong sense, even for first order programs.*

Proof By a reduction from the *Partially Ordered Knapsack problem*, which is defined as follows [17]:

> Given a finite partially ordered set $(U, \preceq)$, for each $u \in U$ a size $s(u) \in \mathcal{Z}^+$ and a value $v(u) \in \mathcal{Z}^+$, and positive integers B and K, is there a subset $U' \subseteq U$ such that if $u \in U'$ and $u' \preceq u$ then $u' \in U'$, and such that $\sum_{u \in U'} s(u) \leq B$ and $\sum_{u \in U'} v(u) \geq K$?

This problem is known to be NP-complete in the strong sense [17]. It is easy to see that the partially ordered knapsack problem is essentially isomorphic to the one-step resource-bounded specialization problem defined above. Suppose we are given an instance of the partially ordered knapsack problem with partial ordered set $(U, \preceq)$, size function s, value function v, and bounds B and K for size and value respectively. The corresponding resource-bounded specialization problem has the set of control points U, with dependence relation between control points given by $\preceq$; the functions wt, cost, and $\mathsf{savings}$ are defined as follows for all $u \in U$:

$$\begin{aligned} \mathsf{wt}(u) &= 1; \\ \mathsf{cost}(u) &= s(u); \\ \mathsf{savings}(u) &= v(u); \end{aligned}$$

Finally, the bounds B and K are the same as in the partially ordered knapsack problem. It is easy to see that this problem admits a solution if and only if the given instance of partially ordered knapsack has a solution, which means that resource-bounded specialization is NP-hard in the strong sense. Membership in NP is straightforward, since we can simply guess a subset $U' \in U$ and verify in polynomial time that it satisfies the closure requirement under $\preceq$, i.e., $u \in U'$ and $u' \preceq$ implies $u' \in U'$, as well as that $\sum_{u \in U'} \mathsf{cost}(u) \le B$ and $\sum_{u \in U'} \mathsf{savings}(u) \ge K$. □

We note in passing that the partially ordered knapsack problem can be solved optimally in pseudo-polynomial time via dynamic programming techniques if $\preceq$ is a "tree" partial order [17]. This means that resource-bounded specialization problem can also be solved optimally in pseudo-polynomial time when the dependence relation between control points is a tree partial order, simply by transforming it into a partially ordered knapsack problem over a set $(U, \preceq)$ where U is the set of control points and $\preceq$ the dependence relation; the size function is $s(u) = \mathsf{cost}(u)$; and the value function is given by $v(u) = \mathsf{wt}(u) \cdot \mathsf{savings}(u)$.

4 A Heuristic Algorithm

Theorem 3.1 implies that the existence of efficient algorithms for optimal one-step resource-bounded program specialization are unlikely. We are forced, therefore, to resort to heuristics. we do this in two phases: first, we determine the costs and benefits associated with specializing each operation that is specializable. We then use this information to choose a set of points to specialize.

Recall, from the discussion in Section 2.1, that for our purposes, a program point needs to associate a set of static environments with a control point, and that this may in some cases preclude static values from being hard-wired into the residual code. It turns out that this may also prevent some specialization from taking place. To see this, consider an expression

$$\mathtt{if}_s(x > 2,\ x + 2,\ x - 3)$$

at a control point with associated set of static environments $\{(x \mapsto 1), (x \mapsto 3)\}$. Even though the conditional here is static, the set of values x can take on does not allow us to determine the outcome of the test uniquely, and therefore prevents us from specializing the test away. On the other hand, if the set of static environments had been $\{(x \mapsto 0), (x \mapsto 1)\}$, the expression would be specializable. This shows that in order to estimate the savings accruing from the specialization of a program point, and the size of the resulting residual code, we need to take into account the set of static environments associated with that point.

As this example suggests, in order to estimate the code size and savings resulting from specialization, we need to be able to determine the set of values an expression can take on given a set of static environments. Let '?' denote a value that is "statically undetermined"—the notion is close to, but not quite the same as, that of a dynamic value, since it is possible to have a static expression whose value, in the context of a set of static expressions, is not uniquely determined: an example is the expression $\mathtt{if}_s(x > 2,\ x + 2,\ x - 3)$

shown above, in the context of the set of static environments $\{(x \mapsto 1), (x \mapsto 3)\}$. Given an expression e and a (single) static environment *env*, let the value of e in environment *env* be given by a function

$$value : \mathsf{Expr} \times (\mathsf{Var} \to Val) \to Val \cup \{?\}.$$

This function is defined essentially as one would expect, the only twist being the need to deal with statically undetermined quantities:[2]

$$
\begin{array}{l}
value([\![c]\!], \rho) = c \\
value([\![x_s]\!], \rho) = lookup(x, \rho) \\
value([\![x_d]\!], \rho) = \,? \\
value([\![op_s(e_1, \ldots, e_n)]\!], \rho) = \textbf{let } u_1 = value(e_1, \rho), \ldots, u_n = value(e_n, \rho) \textbf{ in} \\
\qquad\qquad \textbf{if } u_i = \,? \text{ for some } i,\ 1 \le i \le n, \textbf{ then } ? \\
\qquad\qquad \textbf{else } \pi_{op}(u_1, \ldots, u_n) \\
\qquad \text{where } \pi_{op} \text{ is the function associated with } op. \\
value([\![op_d(e_1, \ldots, e_n)]\!], \rho) = \,? \\
value([\![\mathtt{if}_s(e_1,\ e_2,\ e_3)]\!], \rho) = \\
\qquad \textbf{if } value(e_1, \rho) = true \textbf{ then } value(e_2, \rho) \\
\qquad \textbf{else if } value(e_1, \rho) = false \textbf{ then } value(e_3, \rho) \\
\qquad \textbf{else } ? \\
value([\![\mathtt{if}_d(e_1,\ e_2,\ e_3)]\!], \rho) = \,? \\
value([\![\mathtt{let}_s\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2]\!], \rho) = value(e_2, \rho[x \mapsto value(e_1, \rho)]) \\
value([\![\mathtt{let}_d\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2]\!], \rho) = value(e_2, \rho[x \mapsto value(e_1, \rho)]) \\
value([\![\mathtt{call}_s(f, e_1, \ldots, e_n)]\!], \rho) = value(body(f), \{x_1 \mapsto value(e_1, \rho), \ldots, x_n \mapsto value(e_n, \rho)\}) \\
value([\![\mathtt{call}_d(f, e_1, \ldots, e_n)]\!], \rho) = \,?
\end{array}
$$

Let Val denote the set of denotable values, and $SEnv = \mathcal{P}(\mathsf{Var} \to Val)$ the set of all static environments, where $\mathcal{P}(S)$ denotes the powerset of a set S. The set of possible values for an expression in the context of a set of static environments can now be defined easily:

$$valueset(e, envs) = \{value(e, \rho) \mid \rho \in envs\}$$

Finally, for notational convenience we define the predicate *unique* that specifies whether the value of an expression is uniquely determined in the context of a given set of static environments:

$$unique(e, envs) \quad \Leftrightarrow \quad |valueset(e, envs)| = 1 \textbf{ and } ? \notin valueset(e, envs)$$

4.1 Estimating Costs and Benefits

We first consider how to estimate (an upper bound to) the size of the code resulting from a single specialization step of an annotated expression e. Let this be given by

$$size : \mathsf{Expr} \times SEnv \times \mathbf{Bool} \to \mathcal{N}$$

where $\mathcal{N}$ denotes the set of natural numbers. The Boolean argument is used to ensure that once a function call has been considered as unfolded, calls in its body are not considered to be unfolded: it starts out as

[2]Statically undetermined values may arise even if we consider only static expressions in the context of a single static environment, since it is possible for a static call to have a statically undetermined value in such an environment, depending on the way in which such calls are defined (e.g., see [24], Sec. 5.5.1).

true in the initial call to *size*, but is set to *false* when evaluating the body of an unfolded function call. This gives a conservative approximation to the amount of unfolding that can occur in a single specialization step. Let $\|op\|$ denote the size of the code required to implement a primitive operation *op*. We now consider how these estimates may be computed:

1. Code need not be generated for constants and static variables whose values are uniquely determined in the context of the given set of static environments (except for those occurring in the context of a `lift` operator, which is considered later). If we assume a load-store architecture for simplicity, and conservatively (in order to avoid making assumptions about register allocation) assume that variables are loaded from memory when referenced, we need a single load instruction for a dynamic variable or a static variable whose value is not uniquely determined:

$$size([\![c]\!], envs, \_) = 0 \tag{1}$$
$$size([\![x_s]\!], envs, \_) = \textbf{if } unique(x_s, envs) \textbf{ then } 0 \textbf{ else } 1 \tag{2}$$
$$size([\![x_d]\!], envs, \_) = 1 \tag{3}$$

2. Static base operations that can be evaluated at specialization time do not require any residual code. The code for static base operations that cannot be so evaluated, and for dynamic base operations, consists of the code for evaluating the arguments, together with the code for the operation itself:

$$size([\![op_s(e_1, \ldots, e_n)]\!], envs, \_) = \\ \textbf{if } \bigwedge_{i=1}^{n} unique(e_i, envs) \textbf{ then } 0 \textbf{ else } \|op\| + \textstyle\sum_{i=1}^{n} size([\![e_i]\!], envs, u) \tag{4}$$
$$size([\![op_d(e_1, \ldots, e_n)]\!], envs, u) = \|op\| + \textstyle\sum_{i=1}^{n} size([\![e_i]\!], envs, u) \tag{5}$$

3. If, in the context of the given set of static environments, the outcome of a static conditional can be determined unambiguosly, then the conditional can be specialized, and the residual code is that for the then-part or the else-part, depending on the outcome of the conditional. Otherwise, and for dynamic conditionals, we need to generate code for the test as well as both branches of the conditional:

$$size([\![\texttt{if}_s(e_1,\ e_2,\ e_3)]\!], envs, u) = \\ \textbf{if } valueset(e_1, envs) = \{true\} \textbf{ then } size([\![e_2]\!], envs, u) \\ \textbf{else if } valueset(e_1, envs) = \{false\} \textbf{ then } size([\![e_3]\!], envs, u) \\ \textbf{else } \|\textbf{if}\| + \textstyle\sum_{i=1}^{3} size([\![e_i]\!], envs, u) \tag{6}$$
$$size([\![\texttt{if}_d(e_1,\ e_2,\ e_3)]\!], envs, u) = \|\textbf{if}\| + \textstyle\sum_{i=1}^{3} size([\![e_i]\!], envs, u) \tag{7}$$

4. We need to generate a single load instruction to load the value of an expression `lift(`e`)` (strictly speaking, a value of a type that does not admit a compact representation may require multiple load instructions: for simplicity we ignore this here):

$$size([\![\texttt{lift}(e)]\!], envs, \_) = 1 \tag{8}$$

5. For a static let expression, $\texttt{let}_s\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$, if the value of e_1 can be uniquely determined in the associated set of static environments, then this value can be hard-wired into e_2, so the resulting code size is simply that for e_2; otherwise it is the code size for the subexpressions e_1 and e_2 (in this case the value of x has to be loaded when computing e_2, but this cost is accounted for when we consider occurrences of x within e_2 and so need not be considered separately here):

$$size([\![\texttt{let}_s\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!], envs, u) = \\ size([\![e_2]\!], envs, u) + \textbf{if } unique(e_1, envs) \textbf{ then } 0 \textbf{ else } size([\![e_1]\!], envs, u) \tag{9}$$
$$size([\![\texttt{let}_d\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!], envs, u) = 1 + size([\![e_1]\!], envs, u) + size([\![e_2]\!], envs, u) \tag{10}$$

6. The code for a static function call consists of the code to evaluate the arguments, together with either the code for (a specialized version of) the body if the call is considered to be unfolded, or the code to implement the call operation itself if it is not. By contrast, the code for a dynamic call consists simply of the code to evaluate the arguments, together with the instructions to actually make the call:

$$
\begin{aligned}
&size([\![\mathtt{call}_s(f, e_1, \ldots, e_n)]\!], envs, u) = \\
&\quad \textstyle\sum_{i=1}^{n} size([\![e_i]\!], envs, u) + \\
&\qquad \mathbf{if}\ u\ \mathbf{then}\ size(body(f), envs, false)\ \mathbf{else}\ \|call\| && (11)\\
&size([\![\mathtt{call}_d(f, e_1, \ldots, e_n)]\!], envs, \_) = \|call\| + \textstyle\sum_{i=1}^{n} size([\![e_i]\!], envs, u) && (12)
\end{aligned}
$$

In summary, the function *size* estimating the code size for an expression is given by equations (1)–(12).

Let e_p be the expression at a control point p, and let *envs* be the associated set of static environments, then the cost associated with that point can be expressed as

$$\mathsf{cost}(p) \;= size(e_p, envs, true).$$

Analogously to the above, let $save$: $\mathsf{Expr} \times SEnv \times \mathbf{Bool} \rightarrow \mathcal{N}$ be a function that expresses the savings due to a single specialization step. Intuitively, $save(e, envs, u)$ gives a (lower bound) estimate of the savings resulting from the specialization of an annotated expression e in the context of the set of static environments *envs*, with the truth value u indicating whether or not function calls are to be unfolded. Let $\ll op \gg$ denote the (maximum) number of instructions necessary to execute a primitive operation op in the residual program. Except for the cases involving `lift` and static calls, the reasoning behind these equations are very similar to that for the *size* function, and hence are not repeated. For an expression $\mathtt{lift}(e)$, we determine the savings incurred for the static expression e, but then subtract 1 because we have to execute an instruction at runtime to load the value of this expression. For a static function call, the Boolean argument u determines whether or not the call is actually considered as being unfolded. If it is, the savings includes the cost of making the call, as well as the savings from specializing the unfolded body; otherwise, the savings are simply those from specializing the computation of the actual parameters.

$$
\begin{aligned}
&save([\![c]\!], envs, \_) = 1 && (1)\\
&save([\![x_s]\!], envs, \_) = \mathbf{if}\ unique(x_s, envs)\ \mathbf{then}\ 1\ \mathbf{else}\ 0 && (2)\\
&save([\![x_d]\!], envs, \_) = 0 && (3)\\
&save([\![op_s(e_1, \ldots, e_n)]\!], envs, u) = \\
&\quad \mathbf{if}\ \bigwedge_{i=1}^{n} unique(e_i, envs)\ \mathbf{then}\ \ll op \gg + \textstyle\sum_{i=1}^{n} save([\![e_i]\!], envs, u) \\
&\quad \mathbf{else}\ \textstyle\sum_{i=1}^{n} save([\![e_i]\!], envs, u) && (4)\\
&save([\![op_d(e_1, \ldots, e_n)]\!], envs, u) = \textstyle\sum_{i=1}^{n} save([\![e_i]\!], envs, u) && (5)\\
&save([\![\mathtt{if}_s(e_1,\ e_2,\ e_3)]\!], envs, u) = \\
&\quad \mathbf{if}\ valueset(e_1, envs) = \{true\}\ \mathbf{then}\ \ll \mathbf{if} \gg + save([\![e_1]\!], envs, u) + save([\![e_2]\!], envs, u) \\
&\quad \mathbf{else\ if}\ valueset(e_1, envs) = \{false\}\ \mathbf{then}\ \ll \mathbf{if} \gg + save([\![e_1]\!], envs, u) + save([\![e_3]\!], envs, u) \\
&\quad \mathbf{else}\ \min(save([\![e_2]\!], envs, u), save([\![e_3]\!], envs, u)) && (6)\\
&save([\![\mathtt{if}_d(e_1,\ e_2,\ e_3)]\!], envs, u) = \\
&\quad save([\![e_1]\!], envs, u) + \min(save([\![e_2]\!], envs, u), save([\![e_3]\!], envs, u)) && (7)\\
&save([\![\mathtt{lift}(e)]\!], envs, u) = \max(save([\![e]\!], envs, u) - 1, 0) && (8)\\
&save([\![\mathtt{let}_s\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2]\!], envs, u) = \\
&\quad save([\![e_1]\!], envs, u) + save([\![e_2]\!], envs, u) && (9)
\end{aligned}
$$

$$save([\![\mathtt{let}_d\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2]\!], envs, u) = save([\![e_1]\!], envs, u) + save([\![e_2]\!], envs, u) \tag{10}$$

$$\begin{aligned} &save([\![\mathtt{call}_s(f, e_1, \ldots, e_n)]\!], envs, u) = \\ &\quad \textstyle\sum_{i=1}^{n} save([\![e_i]\!], envs, u) + \\ &\qquad \textbf{if } u \textbf{ then} \ll\!\mathtt{call}\!\gg + \ save(body(f), envs, false) \textbf{ else } 0 \end{aligned} \tag{11}$$

$$save([\![\mathtt{call}_d(f, e_1, \ldots, e_n)]\!], envs, u) = \textstyle\sum_{i=1}^{n} save([\![e_i]\!], envs, u) \tag{12}$$

If the expression occurring at a control point p is e_p, with associated set of static environments $envs$, we can now write

$$\mathsf{savings}(p) \ = save(e_p, envs, true).$$

The equations given have a shortcoming, relating to the handling of static calls, that can sometimes result in too little specialization. The problem is that when estimating the code size for a static call that is unfoldable, the set of static environments with which the size of the body of the called function is estimated is taken to be the same as that at of the static call (equation 11 for *size*). This can be imprecise enough to fail to notice code that can be specialized away after unfolding. This can result in an overestimate of the size of the unfolded body and an underestimate of the associated savings.

The problem can be illustrated by considering the program shown in Section 2.1, rewritten in the abstract syntax of Section 2, together with the set of static environments at each point:

Code	*Static Environments*
$\mathtt{f(y}_d\mathtt{,\ i}_s\mathtt{)\ =}$	$\{(\mathtt{i}_s \mapsto 10000), (\mathtt{i}_s \mapsto 9999), \ldots, (\mathtt{i}_s \mapsto 0)\}$
$\quad\mathtt{if}_s\mathtt{(i}_s\ \mathtt{=}_s\ \mathtt{0,}$	$\{(\mathtt{i}_s \mapsto 10000), (\mathtt{i}_s \mapsto 9999), \ldots, (\mathtt{i}_s \mapsto 0)\}$
$\qquad\mathtt{y}_d\mathtt{,}$	$\{(\mathtt{i}_s \mapsto 0)\}$
$\qquad\mathtt{let\ y1}_d\ \mathtt{=}\ \mathtt{y}_d\ \mathtt{+}_d\ \mathtt{lift(i}_s\mathtt{)\ in}$	$\{(\mathtt{i}_s \mapsto 10000), (\mathtt{i}_s \mapsto 9999), \ldots, (\mathtt{i}_s \mapsto 1)\}$
$\qquad\mathtt{let\ i1}_s\ \mathtt{=}\ \mathtt{i}_s\ \mathtt{-}_s\ \mathtt{1\ in}$	$\{(\mathtt{i}_s \mapsto 10000), (\mathtt{i}_s \mapsto 9999), \ldots, (\mathtt{i}_s \mapsto 1)\}$
$\qquad\quad\mathtt{call}_s\mathtt{(f,\ y1}_d\mathtt{,\ lift(i1}_s\mathtt{))}$	$\{(\mathtt{i1}_s \mapsto 9999), \ldots, (\mathtt{i1}_s \mapsto 0)\}$
$\quad\mathtt{)}$	

To determine the size of the static call, we try to determine the size of the body of `f()` using the set of static environments at the call site, i.e., $\{(\mathtt{i1}_s \mapsto 9999), \ldots, (\mathtt{i1}_s \mapsto 0)\}$. Unfortunately, this set of environments does not allow us to predict the outcome of the test in the static conditional in the body. Because of this, the code size and savings of the unfolded call is estimated very conservatively, with the conditional taken as being unspecialized in the unfolded body. However, if the sets of static environments at different points of the function are recomputed after unfolding the call, we have the following:

Code	*Static Environments*
$\mathtt{f(y}_d\mathtt{,\ i}_s\mathtt{)\ =}$	$\{(\mathtt{i}_s \mapsto 10000), (\mathtt{i}_s \mapsto 9998), \ldots, (\mathtt{i}_s \mapsto 0)\}$
$\quad\mathtt{if}_s\mathtt{(i}_s\ \mathtt{=}_s\ \mathtt{0,}$	$\{(\mathtt{i}_s \mapsto 10000), (\mathtt{i}_s \mapsto 9998), \ldots, (\mathtt{i}_s \mapsto 0)\}$
$\qquad\mathtt{y}_d\mathtt{,}$	$\{(\mathtt{i}_s \mapsto 0)\}$
$\qquad\mathtt{let\ y1}_d\ \mathtt{=}\ \mathtt{y}_d\ \mathtt{+}_d\ \mathtt{lift(i}_s\mathtt{)\ in}$	$\{(\mathtt{i}_s \mapsto 10000), (\mathtt{i}_s \mapsto 9998), \ldots, (\mathtt{i}_s \mapsto 2)\}$
$\qquad\mathtt{let\ i1}_s\ \mathtt{=}\ \mathtt{i}_s\ \mathtt{-}_s\ \mathtt{1\ in}$	$\{(\mathtt{i1}_s \mapsto 9999), (\mathtt{i1}_s \mapsto 9997), \ldots, (\mathtt{i1}_s \mapsto 1)\}$
$\qquad\quad\mathtt{if}_s\mathtt{(i1}_s\ \mathtt{=}_s\ \mathtt{0,}$	$\{(\mathtt{i1}_s \mapsto 9999), (\mathtt{i1}_s \mapsto 9997), \ldots, (\mathtt{i1}_s \mapsto 1)\}$
$\qquad\qquad\mathtt{y1}_d\mathtt{,}$	$\{\ \}$
$\qquad\qquad\mathtt{let\ y2}_d\ \mathtt{=}\ \mathtt{y}_d\ \mathtt{+}\ _d\ \mathtt{lift(i1}_s\mathtt{)\ in}$	$\{(\mathtt{i1}_s \mapsto 9999), (\mathtt{i1}_s \mapsto 9997), \ldots, (\mathtt{i1}_s \mapsto 1)\}$
$\qquad\qquad\mathtt{let\ i2}_s\ \mathtt{=}\ \mathtt{i1}_s\ \mathtt{-}_s\ \mathtt{1\ in}$	$\{(\mathtt{i2}_s \mapsto 9998), (\mathtt{i2}_s \mapsto 9996), \ldots, (\mathtt{i2}_s \mapsto 0)\}$
$\qquad\qquad\quad\mathtt{call}_s\mathtt{(f,\ y2}_d\mathtt{,\ lift(i2}_s\mathtt{))}$	$\{(\mathtt{i2}_s \mapsto 9998), (\mathtt{i}_s \mapsto 9996), \ldots, (\mathtt{i2}_s \mapsto 0)\}$
$\qquad\quad\mathtt{)}$	
$\quad\mathtt{)}$	

Input : A set of control points P together with a dependence relation $\leadsto$ on P ($p \leadsto q$ means p depends on q); functions $\mathsf{wt} : P \to \mathcal{Z}$, $\mathsf{cost} : P \to \mathcal{Z}$, $\mathsf{savings} : P \to \mathcal{Z}$; and a resource bound B.

Output : A set of control points $Q \subseteq P$ that can be specialized without exceeding the resource bound B.

Method :

1. For each $p \in P$ compute, using depth-first traversals,

   $$\begin{aligned}\mathsf{CumCost}(p) &= \sum\{\mathsf{cost}(q) \mid p \leadsto^* q\} \\ \mathsf{CumSvgs}(p) &= \sum\{\mathsf{wt}(q) \cdot \mathsf{savings}(q) \mid p \leadsto^* q\}\end{aligned}$$

2. Let $\mathit{Candidates} = \{p \in P \mid \mathsf{CumCost}(p) \leq B\}$.
   If $\mathit{Candidates} = \emptyset$ then $Q = \emptyset$;
   otherwise, $Q = \{p \mid q \leadsto^* p\}$, where $q \in \mathit{Candidates}$ satisfies:

   (i) $\mathsf{CumSvgs}(q) \geq \mathsf{CumSvgs}(p)$ for all $p \in \mathit{Candidates}$; and

   (ii) $\mathsf{CumSvgs}(p) = \mathsf{CumSvgs}(q)$ implies $\mathsf{CumCost}(q) \leq \mathsf{CumCost}(p)$ for all $p \in \mathit{Candidates}$.

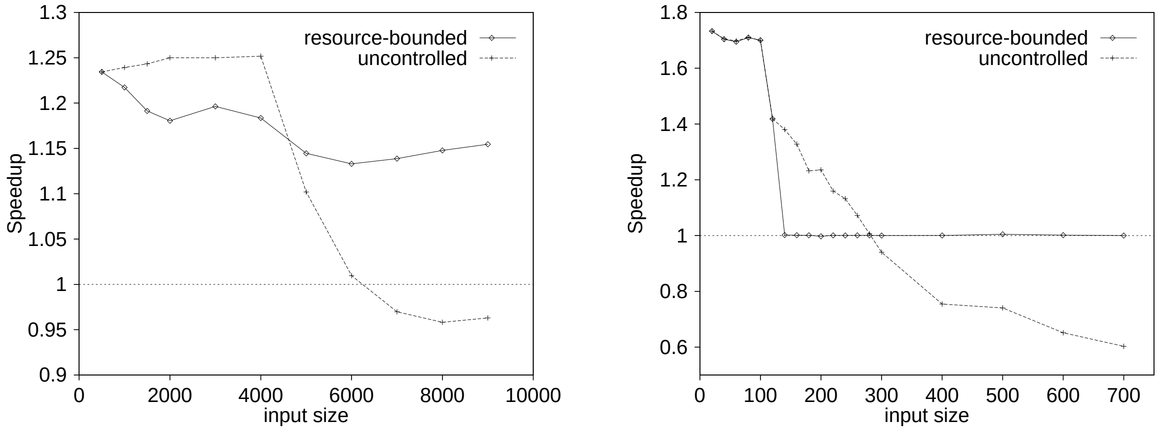
3. Let $Q' = Q \cup \{q' \mid \exists q \in Q : q' \text{ is a static subexpression of } q\}$;
4. return Q'.

Figure 3: A heuristic algorithm for the accountant component

It is clear that the outcome of the static conditional $\mathtt{if}_s\mathtt{(i1}_s\ \mathtt{=}_s\ \mathtt{0,\ \ldots)}$ can now be predicted, and the size and savings determined more accurately. The problem is that if the overestimation of code size due to this reason is high enough, it may prevent the unfolding of a call, in which case we will not be able to discover some opportunities for subsequent specialization: e.g., in the example above, it could be that the code size estimate prior to unfolding is too high to allow unfolding given the available resource bounds, even though the unfolded body after the static conditional has been specialized away may be small enough to fit within these bounds. One possible way to address this problem would be to estimate code sizes of unfolded bodies more carefully, by first determining the set of static environments that would be associated with each point in the unfolded body, then using these static environments to estimate code sizes.

4.2 Cumulative Costs and Benefits

Because of the congruence requirements, it will not be possible, in general, to select operations for specialization in isolation: when a point is selected for specialization, it will be necessary to ensure that all points it depends on are also selected. This means that the total cost of specializing a given point p is given by the cost of p together with the cost of all of the points it depends on. On the other hand, since all of the points that p depends on are also specialized when p is specialized, the total savings resulting from the specialization of p is given by the savings for p together with the savings for all of the points p depends on. We refer to these values as the *cumulative cost* $\mathsf{CumCost}(p)$ and the *cumulative savings* $\mathsf{CumSvgs}(v)$ respectively. Let $p \leadsto q$ denote that point p depends on point q, and let $\leadsto^*$ denote the reflexive transitive closure of $\leadsto$. Then, for any point p, the values of $\mathsf{CumCost}(p)$ and $\mathsf{CumSvgs}(p)$ can be computed using depth-first search to visit each point q such that $p \leadsto^* q$, i.e., the point p itself together with all the points that p depends on, accumulating the costs and savings of each node visited during the traversal. If the program being specialized is of size n (this could be either the original program, or a partly specialized program that is fed back from the specialized to the accountant), there are n control points, and the worst-case complexity of the computation of cumulative costs and savings for all points is $O(n^2)$ time and $O(n)$ space. Notice that (i) it is not correct to write $\mathsf{CumCost}(p) = \mathsf{cost}(p) + \sum\{\mathsf{CumCost}(q) \mid p \leadsto q\}$, since this can sometimes cause the cost of some



(a) convolution (b) matrix multiplication

Figure 4: Resource-Bounded vs. Uncontrolled Specialization

nodes to be counted more than once; and (*ii*) if, instead of repeated depth-first traversals of the dependency graph, we explicitly associate, with each control point, the set of all of the points it depends on, we incur a quadratic space cost, which can be prohibitive for large programs.

Once the cumulative costs and savings corresponding to each control point have been determined, we are in a position to determine the set of points that can be specialized without exceeding the available resources. For this, we simply find a control point with largest cumulative savings whose cumulative cost does not exceed the available resource bounds. This point, and all of its predecessors in the dependency graph, are now marked for specialization. We also mark for specialization each of static subexpressions of the set of expressions so marked, since the costs of these subexpressions were taken into account for when computing the cost of the expressions that were determined to be specializable without exceeding the available resource bounds.

At this point, there may still be enough resources available, after we update our estimate of available resources to account for the control points that have been chosen for specialization, to leave room for further specialization. However, if we simply repeat the above procedure—that is, if we take an available control point whose cumulative savings are the highest among the remaining nodes, and whose resource requirements do not exceed the available resources—we can conceivably consider some nodes twice. The reason is that the cumulative cost of a node a is determined using the cost of all of the nodes it depends on. It may happen that some of these predecessors have already been marked for specialization in the previous step: in this case, the costs and benefits resulting from the specialization of those nodes have already been accounted for. However, the cumulative cost and savings for the node a have not been updated to account for this, which means that they no longer correctly reflect the incremental costs and savings for a given the specialization decisions that have already been made. We could, in principle, rectify this problem by updating the cumulative costs and savings for the remaining nodes appropriately, and then repeat the above procedure, until no further specialization can be carried out. We choose to not do this, since we get essentially the same effect by having the accountant return to the specializer the first set of points it has marked for specialization as described above, then have the specializer query the accountant again with the resulting specialized program. The resulting algorithm is shown in Figure 3.

5 Experimental Results

The ideas described here have not yet been implemented as part of a partial evaluator: the results we describe were obtained via hand-simulation. At this time, we have had time to run experiments on only a few programs. In this section we describe two of the experiments we conducted in order to explain our results and put them in proper perspective. These were run on a lightly-loaded 25 MHz SPARC IPC with 64 Kbytes of cache and 32 Mbytes of main memory.[3] The resource bound used was 16384 (the number of 4-byte words in a 64 Kbyte cache). Runtimes were obtained using the *gettimeofday* system call and taking the smallest time from five runs of each program.

The first experiment involved convolution-like program mentioned in Section 1. This is a simple Scheme program that, given two n-element vectors $\bar{x}$ and $\bar{y}$, computes $\sum_{j=1}^{n} \sum_{i=1}^{n} x_i y_j$:

```
(define (conv x y)
   (define (conv0 x y acc)
      (if (null? y)
          acc
          (conv0 x (cdr y)
                   (conv1 x (car y) acc))))
   )
   (define (conv1 x y0 acc)
      (if (null? x)
          acc
          (conv1 (cdr x) y0
                 (+ acc (* (car x) y0))))
   )
   (conv0 x y 0)
```

Our aim was to specialize the function **conv(x, y)** to the first argument, **x**, and measure the speedups obtained for different lengths of **x**. We used Similix [6] running on the **scm** Scheme interpreter for the specialization, and the Bigloo Scheme-to-C translator (version 1.8) [30] invoked as **bigloo1.8 -O4 -unsafe -farithmetic**, with **gcc** version 2.7.2 as the back-end compiler, to produce executables. It turned out that the memory requirements of Similix and Bigloo were higher than we could afford: the **scm** interpreter running Similix ran out of memory for an input length of 300, while Bigloo was unable to compile the specialized program corresponding to $n = 200$ due to a stack overflow in the preprocessor phase. Fortunately, the structure of both the Scheme code generated by Similix, and the resulting C code obtained from Bigloo, were sufficiently regular in this case as to allow us to extrapolate to larger values of n. Our performance numbers were thus obtained by extrapolating in this manner, using a script to generate C code very similar to what would have been generated given infinite memory. Some further modifications, such as flattening of deeply nested expressions, were necessary to allow **gcc** to parse the input without exhausting the parser stack; we were careful to ensure that none of these transformations changed the essential characteristics of the computation. We then compiled this code using **gcc -O** and timed the resulting executables.

Our second program was a straightforward nested-loops floating-point matrix multiplication program, specialized to one of the matrices. For this, we used a C program, with matrices represented in the natural C style, i.e., as an array of arrays of floating point values. The specialized versions in this case were generated directly using a script, as in the previous case discussed above, and compiled with **gcc -O2**.

The performance of the resulting programs is shown in Figure 4. Figure 4(a) shows the performance of the convolution program. Here, because the resource-bounded specialization is conservative in its estimate of the size of the inner loop of the program, it stops unrolling the loop "too soon," resulting in an early drop in performance for resource-bounded specialization compared to that of uncontrolled specialization. For larger input sizes, however, resource-bounded specialization maintains its speedup while speedups for uncontrolled specialization drops off quickly once the inner loop can no longer fit in the instruction cache. Overall, the code resulting from resource-bounded specialization maintains a speedup of about 15%–20%

[3] We chose this architecture, despite the fact that it is somewhat dated, because the absence of multi-level caches simplified the memory hierarchy and made for a better fit with our current cost model.

over the unspecialized code, while that resulting from uncontrolled specialization is initially about 25% faster than the unspecialized code, but ends up about 5% slower. The performance of the matrix multiplication program is shown in Figure 4(b). In this case, it turns out that the for matrices larger than 128×128, the single combined cache cannot accommodate all of the data as well as the code—this accounts for the steep performance drop for both the uncontrolled and the resource-bounded specialized versions at about this input size. At this point (actually slightly earlier, because of its conservative data reference estimates), resource-bounded specialization stops code specialization because it determines that all of the available resources have been exhausted; as a result, the resource-bounded specialized code is identical to the unspecialized code for larger inputs, and the speedup is 1.0 (since this case worked directly with C arrays, it did not get the small speedups resulting from specialization of `CAR()` and `CDR()` operations as in the convolution program). It can be seen, however, that in either case, by avoiding uncontrolled code growth, resource-bounded specialization is able to avoid the dramatic performance loss suffered by uncontrolled specialization for large inputs.

5.1 Discussion of Performance Results

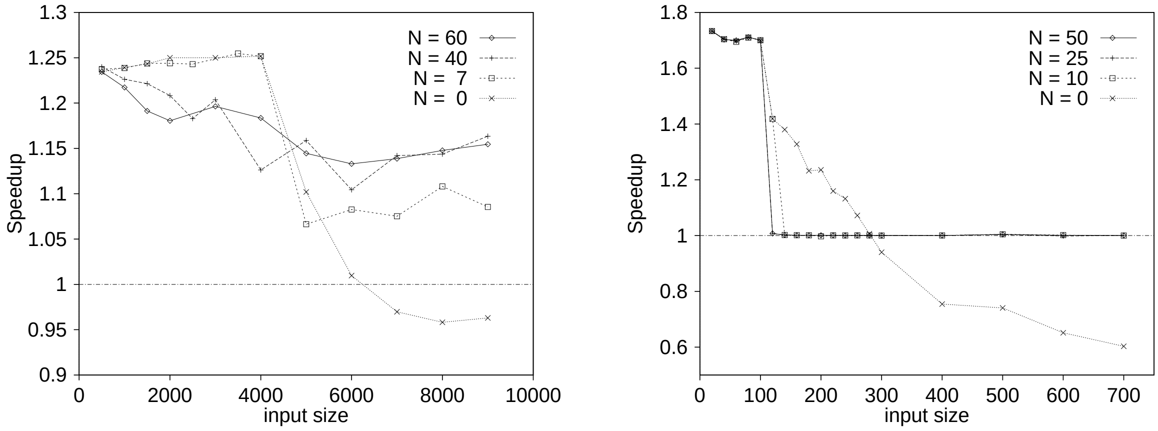
In order to fully understand the behavior illustrated in Figure 4, it is important to consider the code being generated for the various programs that we ran. First, consider the convolution program. The C code generated by Bigloo for the inner loop of the original unspecialized program has the following form (here `NULLP()`, `CINT()` and `CDR()` are Bigloo macros with the behavior one intuitively expects):

```
       acc_0 = 0;
loop:  if (NULLP(x)) acc = acc_0;
       else {
         acc_0 += (long)CINT(CAR(x))*(long)CINT(y);
         x = CDR(x);
         goto loop;
       }
```

By contrast, in the program specialized by Similix without regard to resource bounds, the inner loop is completely unrolled, resulting in C code of the form:

```
       acc_0 = 0;
       acc_0 += 1000*(long)CINT(y);
       acc_0 +=  999*(long)CINT(y);
       acc_0 +=  998*(long)CINT(y);
          ...
       acc_0 +=    3*(long)CINT(y);
       acc_0 +=    2*(long)CINT(y);
       acc_0 +=    1*(long)CINT(y);
       acc = acc_0;
```

Finally, consider the code produced by resource-bounded specialization. For this program, this code consists of a loop, obtained by partially unrolling the inner loop of the original program, whose body does not exceed the amount of available cache, followed by a straight-line code segment corresponding to any left-over computations. A first observation is that, unlike the fully-unrolled version, the assignments to `acc_0` inside the loop in this program cannot have the value of the static data hard-wired into them. However, it is not necessary to give up all hope of specializing these statements and revert to explicitly accessing the elements of a list using `CAR()` and `CDR()` operations as in the unspecialized program: the specializer knows the actual values of the static data, and the sequence in which they are accessed, and it is not unreasonable to suppose that it can eliminate some of the overhead of accessing these values by replacing the list by an array of integers that is accessed via a pointer. The resulting code therefore has the following structure:



(a) convolution (b) matrix multiplication

Figure 5: Speedups for Different Estimates of Inner Loop Size

```
long tbl[] = {1000, 999, 998, ..., 3, 2, 1};
  ...
acc_0 = 0;
ptr = &(tbl[0]);
cnt = ...     /* no. of iterations of the
                 partially unrolled loop */
for (i = 0; i < cnt; i++) {
             /* partially unrolled loop */
  acc_0 += (*ptr++)*(long)CINT(y);
  acc_0 += (*ptr++)*(long)CINT(y);
     ...
  acc_0 += (*ptr++)*(long)CINT(y);
}
/*-- remainder of computation --*/
acc_0 += 74*(long)CINT(y);
acc_0 += 73*(long)CINT(y);
   ...
acc_0 +=  3*(long)CINT(y);
acc_0 +=  2*(long)CINT(y);
acc_0 +=  1*(long)CINT(y);
acc = acc_0;
```

There are two low-level aspects of the particular machine on which our experiments were carried out that have profound effects on the performance characteristics of these programs. The first is that the SPARC IPC does not have a hardware multiplier, so integer multiplication is carried out in software via a call to a function (implemented in hand-coded assembly code) whose body contains about 50 instructions. The second is that when one of the two operands of an integer multiplication operation is a constant, C compilers such as `cc` and `gcc` are able to implement the multiplication using a sequence of bit-manipulation operations such as left-shift, add, and logical-or, which turn out to be considerably cheaper than a call to the general-purpose multiplication function. As a result, for an input length of 1000, for example, the specialized program obtained from Similix, with its fully unrolled loop, is about 9.5 times faster than the original program—considerably more than a straightforward examination of the C source code would suggest. In the partially unrolled loops resulting from resource-bounded specialization, unfortunately, the expressions involving multiplication do not have integer constants hard-wired into them as operands; as a result, these operations are implemented using calls to the multiplication function. Because of this, for small input sizes, the code resulting from the resource-bounded specialization turn out to be roughly an order of magnitude slower than the fully unrolled versions.

It can be argued, however, that speed improvements resulting from C compiler tricks for a particular operation should not be counted towards the gains resulting from partial evaluation. To see this, suppose that instead of integer multiplication, the "product operation" in this computation was floating point multiplication or bitwise-xor: the programs resulting from partial evaluation would have been essentially identical in each case, modulo the change in the product operator, but their relative performance would have been considerably different because of the absence of corresponding bit-twiddling tricks in the C compiler. Similarly, if the processor had a hardware multiplier, the cost of carrying out the multiplication operations would be reduced considerably, again leading to significant differences in relative performance.

For these reasons, we felt that in order to obtain a fair comparison between resource-bounded and uncontrolled partial evaluation, we should separate out effects due to low-level compiler tricks, by forcing the C compiler to always use the general purpose multiplication function for integer multiplication. It turns out that in this case, the fully unrolled loop resulting from uncontrolled specialization of the convolution program is about 25% faster than the unspecialized program for small input sizes (this does not affect the matrix multiplication example, since this program does not contain any integer multiplication operations). The speedups resulting from these programs, for different input lengths, is shown in Figure 4.

Since partial evaluation is usually formulated as a source-to-source transformation, a question of some interest is: how precise does the resource-bounded specializer's estimates of the size of the generated code have to be? To examine this issue, we examined the performance due to resource-bounded specialization of the convolution program for different estimates of the size of the inner loop. The results are shown in Figure 5, where the parameter N denotes the specializer's estimate of the number of instructions in the inner loop of the program.[4] We found that as long as the size estimate is conservative enough to ensure that the specialized code does not overflow the cache, the performance for different values of N is noticeable but not huge. For example, for the convolution program, shown in Figure 5(a), the difference between the curve for $N = 40$ and that for $N = 60$ is typically about 5%–7%. The reason the performance curve for $N = 7$—which is actually the closest to the actual size of the generated code—drops steeply at an input size of 4000 is that once the input exceeds this size, the resource-bounded specializer determines that the inner loop should not be unrolled further, and switches to pointer-based accesses of data from a table, which results in a memory reference with an additional level of indirection for each data value (however, unlike uncontrolled specialization, this halts further performance degradation). The difference between the curves for $N = 7$ and $N = 60$ are again not intolerably large. For the matrix multiplication program, shown in Figure 5(b), going from $N = 10$ to $N = 50$ has only a very minor effect on the overall speedups. This leads us to believe that it is possible to use resource-bounded specialization to attain reasonable performance as long as the specializer makes conservative but reasonable assumptions about the compiler technology being used to generate the executable code.

The discussion thus far has focused on a single resource bound. We believe that in general, it will be necessary to model the memory hierarchy of a computational environment in more detail: for example, many processors now come equipped with two (and, occasionally, three) levels of cache memory, and it is not unreasonable to consider hardware registers as another level above all of these. Whether or not a certain increase in code size is worthwhile then depends greatly on the relative costs of accessing the different levels of such a multi-level hierarchy. We hope to address such issues in future work.

[4] The reason Figure 4(a) shows the graph corresponding to $N = 60$ is that our code size estimates were not smart enough to realize that multiple calls to the integer multiplication routine within an unrolled loop would nevertheless result in a single copy of the code for that function being resident in the cache. In other words, it assumed—conservatively—that each such call would be unfolded.

6 Discussion

The introduction of an "accountant" that guides specialization decisions based on resource usage can lead to some interesting generalizations to ideas traditionally used in offline partial evaluation. Here we explore some of these.

6.1 Termination Considerations

In traditional offline partial evaluation, since the specializer blindly specializes all computations annotated as "static", the responsibility for ensuring termination falls on the binding time analysis [3, 18, 21]. This is undesirable, both for conceptual and pragmatic reasons. Conceptually, it mixes two independent concerns: the question of what *can* be specialized, and that of what *should* be specialized. In other words, traditional offline partial evaluation overloads the binding time annotation "dynamic" to mean both "cannot be statically computed" and "should not be (blindly) computed statically due to termination concerns." Pragmatically, it means that some kinds of transformations are necessarily ruled out, even though they may lead to performance improvements: for example, a recursive function whose recursion is controlled by a dynamic value will not be unfolded, even though a limited amount of unfolding could be beneficial in improving program performance.

In our model, by contrast, the separation of concerns is much sharper: the binding time analysis is concerned solely with identifying which computations can be specialized, while accountant is responsible for deciding which computations should be specialized. As mentioned at the end of Section 4.2, the number of invocations of the accountant—and, therefore, the number of specialization steps that take place—depends on, among other things, the rate at which the amount of available resources decreases as the program is specialized. It can happen that the amount of code growth during a specialization step due to unfolding is exactly equal to the amount of code that is then specialized away, resulting in no net change in either code size or resource usage: this can lead to nontermination of partial evaluation in programs with static infinite loops. For this reason, while most of the discussion thus far has focused on effective utilization of resources by the specialized code, it may not be unreasonable to require the accountant to be responsible for termination of partial evaluation as well. If this is done, the model described in this paper can be extended to obtain some additional flexibility during partial evaluation. For example, in a program that spends much of its time in a recursive function whose recursion is controlled by a dynamic parameter, it may be possible to allow this function to be unfolded to a limited extent, and this can have beneficial performance effects without compromising termination of partial evaluation.

6.2 Flexible Binding of Resource Information

Hard-wiring in resource information pertinent to a particular computational environment too early in the specialization process may lead to an overly inflexible system. This potential inflexibility can be handled by considering the resource information to be an additional static input whose value becomes available at some point during a multi-level specialization process [19]. By appropriately choosing the level at which the resource information becomes available, we can obtain a variety of behaviors: for example, by making the resource information known at an early level, we can get a partial evaluator that can handle different programs for a particular machine, while by making the resource information available late in the multi-level specialization process we can get a generating extension for a particular program that can be used on a variety of different machines.

6.3 Value-selective Specialization using "The Trick"

A standard technique for binding-time improvement for variables of bounded static variation is "the trick" [24]. The basic idea is as follows: suppose we have the following program fragment from a network communication protocol:

```
process(status, pkt)
```

where `status` is the status of the previous transmission, and `pkt` a packet to be sent. Suppose that `status` is dynamic, but can take on values only from the set {*ok*, *timeout*, *corrupt*}. Then we can rewrite the above computation as (something semantically equivalent to):

```
case status of
   ok:      process(ok, pkt);
   timeout: process(timeout, pkt);
   corrupt: process(corrupt, pkt);
end
```

Specialization of this rewritten program specializes each of the calls to `process()` to the value of the corresponding first argument, as desired. Now suppose that it turns out that in the vast majority of cases the value taken on by `status` is *ok*. In this case, it may make sense to generate specialized code for this case only (especially if, as is very often the case, the rarely-executed exception handling code is large and bulky), and resort to the general-purpose unspecialized code for the other cases.[5] Since resource-bounded specialization takes execution weights into account when determining the cumulative savings for various control points, it can achieve a similar effect, specializing for only those values that yield sufficient benefits without exceeding the available resources. Of course, a similar effect can be obtained by manually writing the code as

```
if (status = ok) then
   process(ok, pkt)
else
   process(status, pkt)
```

However, resource-bounded specialization can offer greater flexibility: for example, continuing the network protocol application line, consider a packet classifier that takes a packet received from the network, identifies which protocol it belongs to, and processes it accordingly. In Europe, such a classifier might find that the X.25 protocol is very commonly used, while in the USA the IP protocol might be found to be much more common. In either case, it makes sense to specialize the code in the packet classifier for the more commonly encountered protocol(s), but this is awkward at best using manual rewriting. With resource-bounded specialization, classifiers in different operational environments can be specialized in different ways without excessive manual intervention.

7 Conclusions

Traditional off-line partial evaluators generally do not take into account the availability of machine resources during specialization. This can adversely affect performance, in extreme cases causing a specialized program to run more slowly than the unspecialized version. In this paper we consider how resource availability considerations can be incorporated into a partial evaluator. We show that optimal resource-bounded specialization is an NP-complete problem, and discuss simple heuristics that can be used to address the problem in practice, and discuss how awareness of resource availability can lead to some interesting generalizations of ideas traditionally used in offline partial evaluation. While our algorithms have not been incorporated into a partial evaluator, preliminary experiments appear encouraging.

Acknowledgements

This paper has benefited greatly from comments by Peter Holst Andersen as well as the anonymous referees.

[5] In operating systems parlance, this kind of selective specialization is referred to as "outlining" [8, 26, 27].

References

[1] L. O. Anderson, "Program Analysis and Specialization for the C Programming Language", DIKU Report No. 94/19, Dept. of Computer Science, University of Copenhagen, 1994.

[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 Technical Report YALEU/DCS/RR-909, Department of Computer Science, Yale University, New Haven, CT.)

[3] P. H. Andersen and C. K. Holst, "Termination Analysis for Offline Partial Evaluation of a Higher Order Programming Language", *Proc. Third International Static Analysis Symposium*, 1996.

[4] R. Baier, R. Glück, and R. Zöchling, "Partial Evaluation of Numerical Programs in Fortran", *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 1994, pp. 119–132. Report 94/9, Dept. of Computer Science, University of Melbourne.

[5] L. Birkedal and M. Welinder, "Partial Evaluation of Standard ML", DIKU Report No. 93/22, Dept. of Computer Science, University of Copenhagen, 1993.

[6] A. Bondorf, *Similix 5.0 Manual*, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, May 1993.

[7] R. M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs", *Journal of the ACM* vol. 24 no. 1, Jan. 1977, pp. 44-67.

[8] C. Castelluccia, "Automating Header Prediction", *Proc. Workshop on Compiler Support for System Software*, Tucson, Feb. 1996, pp. 44–53.

[9] W. Y. Chen, P. P. Chung, T. M. Conte, and W. W. Hwu, "The Effect of Code Expanding Optimizations on Instruction Cache Design", *IEEE Transactions on Computers* 42(9), Sept. 1993, pp. 1045–1057.

[10] C. Consel, "Binding Time Analysis for Higher Order Untyped Functional Languages", *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pp. 264–272.

[11] O. Danvy, N. Heintze and K. Malmkjær, "Resource-Bounded Partial Evaluation", *ACM Computing Surveys* vol. 28 no. 2, June 1996, pp. 329–332.

[12] J. W. Davidson and A. M. Holler, "Subprogram Inlining: A Study of its Effects on Program Execution Time", *IEEE Transactions on Software Engineering* vol. 18 no. 2, Feb. 1992, pp. 89–102.

[13] J. W. Davidson and S. Jinturkar, "An Aggressive Approach to Loop Unrolling", *Proc. Compiler Construction '96*.

[14] J. Dean and C. Chambers, "Towards Better Inlining Decisions using Inlining Trials", *Proc. 1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, June 1994, pp. 273–282.

[15] J. Dean, C. Chambers and D. Grove, "Selective Specialization for Object-Oriented Languages", *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995, pp. 93–102.

[16] J. Dongarra and A. R. Hinds, "Unrolling Loops in FORTRAN", *Software Practice and Experience* vol. 9 no. 3, March 1979, pp. 219–226.

[17] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

[18] A. J. Glenstrup and N. D. Jones, "BTA Algorithms to Ensure Termination of Off-line Partial Evaluation", in *Perspectives of System Informatics: Proceedings of the Andrei Ershov Second International Memorial Conference*, June 1996.

[19] R. Glück and J. Jørgensen, "Efficient Multi-Level Generating Extensions for Program Specialization", *Proc. International Symposium on Programming Languages, Implementation, Logics and Programs (PLILP)*, 1995.

[20] C. Gurr, *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*, Ph.D. Thesis, University of Bristol, 1994.

[21] C. K. Holst, "Finiteness Analysis", *Proc. Functional Programming and Computer Architecture*, 1991, pp. 473–495.

[22] W. W. Hwu and P. H. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler", *Proc. 16th. International Symposium on Computer Architecture*, May 1989, pp. 242–251.

[23] W. W. Hwu and P. H. Chang, "Inline Function Expansion for Compiling C Programs", *Proc. SIGPLAN '89 Conference on Programming Language Design and Implementation*, June 1989, pp. 246–257.

[24] N. D. Jones, C. K. Gomard and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice Hall, 1993.

[25] S. McFarling, *Program Analysis and Optimization for Machines with Instruction Cache*, Ph.D. Dissertation, Stanford University, Sept. 1991.

[26] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley, "Improving the I-Cache Effectiveness of Network Software", *Proc. Workshop on Compiler Support for System Software*, Tucson, Feb. 1996, pp. 29–36.

[27] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley, "Analysis of Techniques to Improve Protocol Processing Latency", *Proc. SIGCOMM '96*, pp. 73–84, Sept. 1996.

[28] K. Pettis and R. C. Hansen, "Profile Guided Code Positioning", *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation*, 1990, pp. 16–27.

[29] D. Sahlin, *An Automatic Partial Evaluator for Full Prolog*, Ph.D. Thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden, 1991. Report TRITA-TCS-9101.

[30] M. Serrano and P. Weis, "Bigloo: a portable and optimizing compiler for strict functional languages" *Proc. Static Analysis Symposium (SAS '95)*, 1995, pp. 366–381.

[31] S. Weiss and J. E. Smith, "A Study of Scalar Compilation Techniques for Pipelined Supercomputers", *Proc. Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)* , Oct. 1987, pp. 105–109.