Resource-Bounded Partial Evaluation [*]

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

Technical Report 96-19
November 1996

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 availability 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.

[*] This work was supported in part by the National Science Foundation under grant CCR-9502826

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, 13, 19]. A central concern guiding these implementations has been to ensure that computations in an input program that can be specialized 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 are, by and large, fairly successful 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, and constrained only by concerns regarding termination, partial evaluators typically ignore the eviailability of machine-level resources, such as registers or cache, when making specialization decisions. The resource contention resulting from such aggressive specialization 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 $\sum_{i=1}^{n} \sum_{j=1}^{n} x_i y_j$ 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.

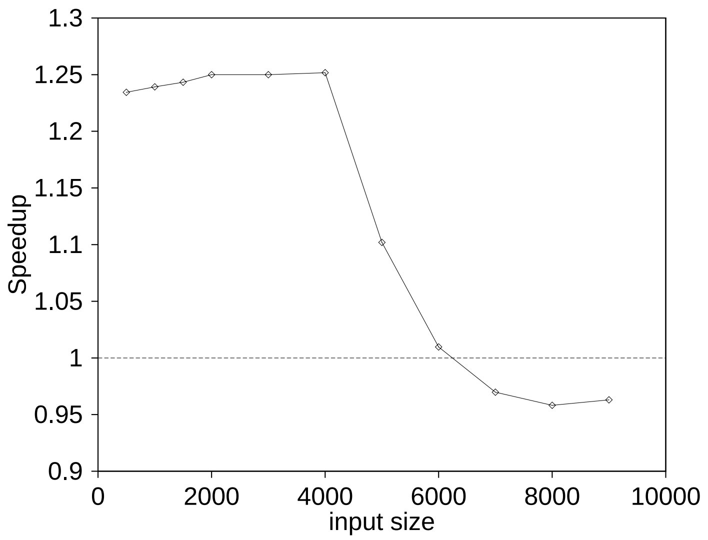

Figure 1: Relative Performance of Specialized Convolution Programs

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

[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 [20], invoked as `bigloo1.8 -O4 -unsafe -farithmetic` 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.

size of the body of the main loop—where most of the execution time of the program is spent—grows linearly as n. While this may appear to be a modest growth rate, it 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 [15]), 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 that are "worth specializing", i.e., 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.* [9], who sketch possible approaches to the problem at a very high level but offer few details.

2 Underlying Concepts

2.1 Program Points

We assume that the programs under consideration for specialization are expressed in a (untyped) first-order functional language. The specific details of syntax are not very important in this discussion, and we will informally use a first-order subset of Scheme for our purposes. Let a *control point* refer to any executable construct within a program (i.e., a node in the program's abstract syntax tree), 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" is taken to be a pair $(cp, senv)$, where cp is a control point and $senv$ a static environment for that point. For our purposes, we need to extend this so that a program point is a pair $(cp, SEnv)$ where $SEnv$ is a *set* of static environments corresponding to the control point cp. 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 100000)
)
```

The sort of code we would intuitively like to generate is that obtained by unrolling this loop, i.e., 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 1000)
 )
```

However, if we take a program point to be a (control point, static environment) pair, then specialization will proceed with the sequence of program points $(L, [\texttt{i} \mapsto 100000]), (L, [\texttt{i} \mapsto 9999]), \ldots$, where L denotes the control point corresponding to the statement `y = 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 associate a set of static environments with each control point: in the special case where this set is a singleton, the values of the static variables can be substituted for the variables, as before.

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 with a binding-time annotated program, which indicates which program point *can* be specialized, together with an estimate of the available resources. The accountant uses the resource information to determine which operations *should* be

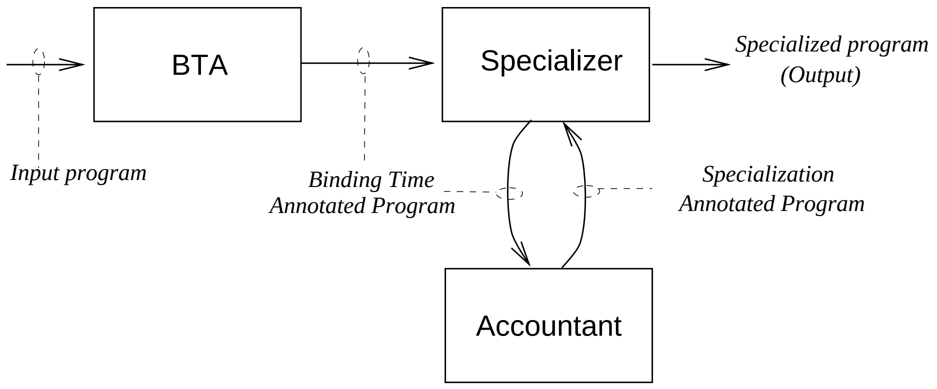
BTA
Specializer
Accountant
Input program
Binding Time Annotated Program
Specialization Annotated Program
Specialized program (Output)

Figure 2: Organization of Resource-Bounded Partial Evaluation

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: we will maintain the distinction between *static/dynamic* annotations made by the binding time analyzer and the *specialize/don't-specialize* annotations made by the accountant for conceptual clarity). The resulting specialization-annotated program is returned to the specializer, which carries out a single specialization step based on the decisions of the accountant and updates its resource availability estimates accordingly. The resulting program, with any new program points annotated with binding time annotations inherited from the original binding-time annotated program where necessary, is used to query the accountant again, together with the updated resource estimates. This process continues 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 program point is annotated for specialization. In some ways, this resembles the notion of "mixline" partial evaluation [8], 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. The specialization-annotated program must satisfy a similar condition. To see this, consider the following program fragment:

```
(define (foo x)
   (define (f x) (+ x (g x) (g (- x))))
   (define (g y) (+ y (h y)))
   (define (h z) ... )
```

```
        (+ x (f 1))
    )
```

The function `h` can be called with the arguments 1 or -1. Suppose we want to specialize `h()` for the argument value 1. Because resource-bounded specialization specializes some, but not necessarily all, of the computations that can be specialized, we have to make sure that the specialized version of `h()` can never be called from a context where its first argument is not 1. In other words, it is necessary to specialize the appropriate call sites for `h()` as well; this, in turn, implies a need to also specialize the function `g()`. In general, we have to ensure that if we choose to specialize a particular computation, then we should also specialize any computation that it depends on, so as to make sure that a specialized version of an operation is not invoked incorrectly.

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 program 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 program 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 for any given program such that the total savings is maximized subject to the constraint that the size of the residual program at the end of the specialization process 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 program points P; functions $\mathsf{wt} : P \rightarrow \mathcal{Z}$, $\mathsf{cost} : P \rightarrow \mathcal{Z}$ and $\mathsf{savings} : P \rightarrow \mathcal{Z}$; and positive integers B, K, is there a set of program 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, even for first order programs.*

Proof By a straightforward reduction from the Knapsack problem, which is known to be NP-complete [16]. □

It turns out that the reduction from the knapsack problem makes no use of the congruence requirement that specialization annotations are required to satisfy. As a result, while the knapsack problem is solvable in pseudo-polynomial time via dynamic programming, it is not clear whether similar techniques are applicable to resource-bounded specialization in the general case, where congruence requirements have to be met. In fact, for higher order programs, it is straightforward to take advantage of the congruence requirement and adapt the reduction for Theorem 3.1 to one where the reduction is from the Partially Ordered Knapsack problem, which is known to be NP-complete in the strong sense [10]. It follows that resource-bounded specialization for higher order programs is NP-complete in the strong sense.

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 program point. We then use this information to choose a set of points to specialize. To simplify the discussion in this section, we assume that instead of passing a binding-time annotated program to the accountant, the specializer passes a set of program points, together with a dependence relation on them that specifies, for any given program point, the set of program points it depends on. This is straightforward to implement, since the set of program points in question are simply those that are marked static, and the dependence relation can be collected during binding time analysis. It is also straightforward to adapt the algorithm given below to work on a binding-time annotated program.

Because of the congruence requirements, it will not be possible, in general, to select program points for specialization in isolation: when a program point is selected for specialization, it will be

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

Output : A set of program 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,

   $\mathsf{CumCost}(p) = \sum\{\mathsf{cost}(q) \mid p \leadsto^* q\}$
   $\mathsf{CumSvgs}(p) = \sum\{\mathsf{wt}(q) \cdot \mathsf{savings}(q) \mid p \leadsto^* q\}$
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:
   (a) for all $p \in \mathit{Candidates}$, $\mathsf{CumSvgs}(q) \geq \mathsf{CumSvgs}(p)$; and
   (b) for all $p \in \mathit{Candidates}$, $\mathsf{CumSvgs}(p) = \mathsf{CumSvgs}(q)$ implies $\mathsf{CumCost}(q) \leq \mathsf{CumCost}(p)$.
3. return Q.

Figure 3: A heuristic algorithm for resource-bounded partial evaluation

necessary to ensure that all points it depends on are also selected. This means that the total cost of specializing a given program 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 program point p, the values of $\mathsf{CumCost}(p)$ and $\mathsf{CumSvgs}(p)$ can be computed using depth-first search to visit each program 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 there are n program points, the worst-case complexity of the computation of cumulative costs and savings for all program 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 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 program 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 program point have been determined, we are in a position to determine the set of program points that can be specialized without exceeding the available resources. For this, we simply find a program 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.

At this point, there may still be enough resources available, after we update our estimate of available resources to account for the program 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 program 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 n 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 n have not been updated to account for this, which means that they no longer correctly reflect the incremental costs and savings for n 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 program 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.

Overall, the worst-case complexity of this algorithm is as follows: $O(n^2)$ time and $O(n)$ space to determine the cumulative cost and savings for each program point; $O(n)$ time and $O(1)$ space to find a suitable program point to specialize (together with the points it depends on); and $O(n)$ time and $O(n)$ space to mark these points for specialization. The overall complexity, therefore, is $O(n^2)$ time and $O(n)$ space.

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 just one program, the convolution-like program discussed in Section 1. In this section we describe the experiments we conducted in order to explain our results and put them in proper perspective.

The program under consideration is a simple Scheme program that, given two n-element vectors $\bar{x}$ and $\bar{y}$, computes $\sum_{i=1}^{n} \sum_{j=1}^{n} x_i y_j$:

```
(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) [20], with `gcc`

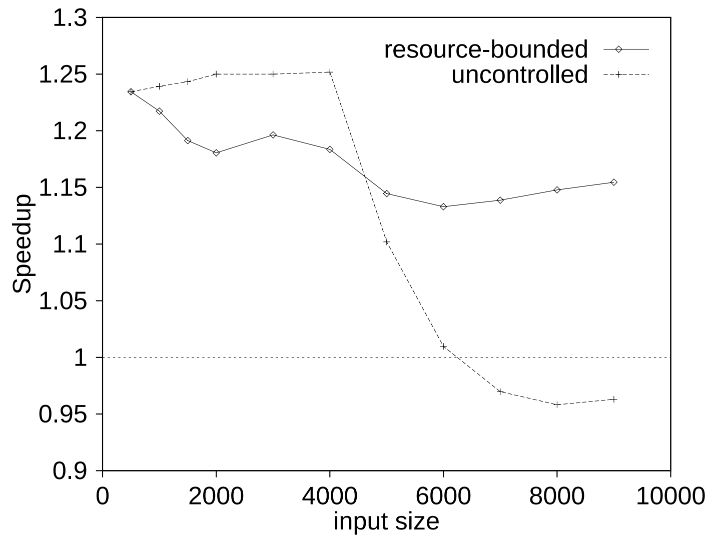

Figure 4: Resource-Bounded vs. Uncontrolled Specialization (Convolution)

version 2.7.2 as the back-end compiler, to produce executables. These were run on a 25 MHz SPARC IPC with 64 Kbytes of cache and 32 Mbytes of main memory. 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 invoked `gcc` on this code exactly as Bigloo would, and timed the resulting executables.

The performance of the resulting programs is shown in Figure 4. In order to understand these curves in detail, it is important to consider the code being generated for the various programs that we ran. 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 whose form is as follows:

```
acc_0 = 0;
if (NULLP(x)) acc = acc_0;
else {
  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:

```
long tbl[] = {1000, 999, 998, ..., 3, 2, 1};
  ...
acc_0 = 0;
if (NULLP(x)) acc = acc_0;
else {
  ptr = &(tbl[0]);
  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);
  }
  acc_0 += 74*(long)CINT(y);         /* remainder of computation */
  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. We decided to do this by forcing the C compiler to always use the general purpose multiplication function. It turned out that this could be done simply by rewriting expressions involving multiplication to avoid a constant operand, e.g., by rewriting a statement '`acc_0 += 937*(long)CINT(y)`' to

```
    tmp = 937; acc_0 += tmp*(long)CINT(y);
```

It turns out that in this case, the fully unrolled loop resulting from uncontrolled specialization is about 25% faster than the unspecialized program for small input sizes.

The speedups resulting from these programs, for different input lengths, is shown in Figure 4. 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% 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. Thus, by avoiding uncontrolled code growth, resource-bounded specialization is able to avoid the dramatic performance loss suffered by uncontrolled specialization for large inputs.

Since partial evaluation is usually formulated as a source-to-source transformation, a question

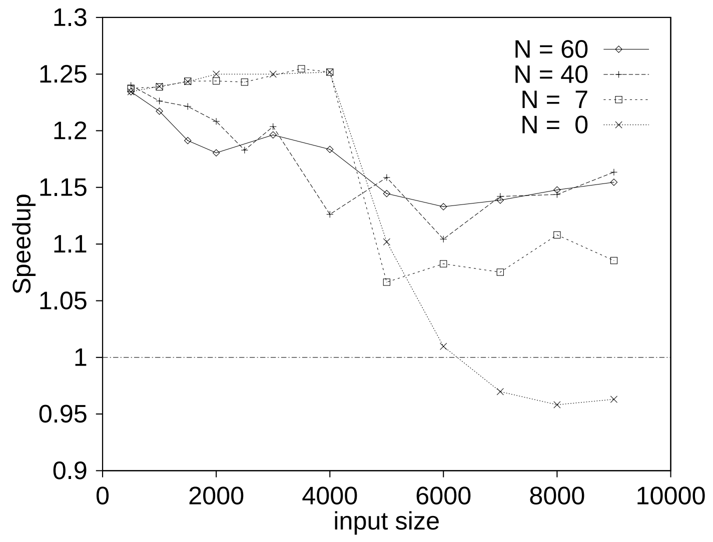

Figure 5: Speedups for Different Estimates of Inner Loop Size (Convolution)

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. 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, 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. 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.

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, 11, 14]. 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. While most of the discussion thus far has focused on effective utilization of resources by the specialized code, it does not seem unreasonable to require the accountant to be responsible for termination of partial evaluation as well. This can result in 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, our approach can 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 [12]. 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" [15]. 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.[2] Since resource-bounded specialization takes execution weights into account when determining the cumulative savings for various program 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.

[2] In operating systems parlance, this kind of selective specialization is referred to as "outlining" [7, 17, 18].

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

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

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

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

[11] 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.

[12] 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.

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

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

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

[16] R, M. Karp, "Reducibility among Combinatorial Problems", in *Complexity of Computer Computations*, eds. R. E. Miller and J. W. Thatcher, Plenum Press, New York, 1972, pp. 85–103.

[17] 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.

[18] 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.

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

[20] M. Serrano, *Bigloo User's Manual*, INRIA Rocquencourt, France, April 1996.