Combining Program and Data Specialization

Sandrine Chirokoff Charles Consel

Compose project, IRISA / INRIA - Université de Rennes 1,
Campus Universitaire de Beaulieu, 35042 Rennes cedex, France.
http://www.irisa.fr/compose

October 1998

Abstract

Program and data specialization have always been studied separately, although they are both aimed at processing early computations. On the one hand, program specialization encodes the result of early computations into a new program and, on the other hand, data specialization encodes the result of early computations in data structures.

In this paper, we present an extension of the Tempo specializer, which performs *both* program and data specialization. We show how these two strategies can be integrated in a unique specializer. This new kind of specializer provides the programmer with complementary strategies which widen the scope of specialization. We illustrate the benefits and limitations of these strategies and their *combination* on a variety of programs.

1 Introduction

Program and data specialization are both aimed at performing computations which depend on early values. However, they differ in the way the result of early computations are encoded: on the one hand, program specialization encodes these results in a residual program, and on the other hand, data specialization encodes these results in data structures.

More precisely, program specialization performs a computation when it only relies on early data, and inserts the textual representation of its result in the residual program when it is surrounded by computations depending on late values. In essence, it is because a new program is being constructed that early computations can be encoded in it. Furthermore, because a new program is being constructed it can be pruned, that is, the residual program only corresponds to the control flow which could not be resolved given the available data. As a consequence, program specialization optimizes the control flow since fewer control decisions need to be taken. However, because it requires a new program to be constructed, program specialization can lead to code explosion if the size of the specialization values is large. For example, this situation can occur when a loop needs to be unrolled and the number of iterations is high. Not only does code explosion cause code size problems, but it also degrades the execution time of the specialized program dramatically because of instruction cache misses.

The dual notion to specializing programs is specializing data. This strategy consists of splitting the execution of program into two phases. The first phase, called the *loader*, performs the early computations and stores their results in a data structure called a *cache*. Instead of generating a program which contains the textual representation of values, data specialization generates a program to perform the second phase: it only consists of the late computations and is *parameterized* with respect to the result of early computations, that is, the cache. The corresponding program is named the *reader*. Because the reader is parameterized with respect to the cache, it is shared by all specializations. This strategy fundamentally contrasts with program specialization because it decouples the result of early computations and the program which exploits it. As a consequence,

as the size of the specialization problem increases, only the cache parameter increases, not the program. In practice, data specialization can handle problem sizes which are far beyond the reach of program specialization, and thus opens up new opportunities as demonstrated by Knoblock and Ruf for graphics applications [7, 4]. However, data specialization, by definition, does not optimize the control flow: it is limited to performing the early computations which are expensive enough to be worth caching. Because the reader is valid for any cache it is passed, an early control decision leading to a costly early computation needs to be part of the loader as well as the reader: in the loader, it decides whether the costly computation much be cached; in the reader, the control decision determines whether the cache needs to be looked up. In fact, data specialization does not apply to programs whose bottlenecks are limited to control decisions. A typical example of this situation is interpreters for low-level languages: the instruction dispatch is the main target of specialization. For such programs, data specialization can be completely ineffective.

Perhaps the apparent difference in the nature of the opportunities addressed by program and data specialization has led researchers to study these strategies in isolation. As a consequence, no attempt has ever been made to integrate both strategies in a specializer; further, there exist no experimental data to assess the benefits and limitations of these specialization strategies.

In this paper, we study the relationship between program and data specialization with respect to their underlying concepts, their implementation techniques and their applicability. More precisely, we study program and data specialization when they are applied separately, as well as when they are combined (Section 2). Furthermore, we describe how a specializer can integrate both program and data specialization: what components are common to both strategies and what components differ. In practice, we have achieved this integration by extending a program specializer, named Tempo, with the phases needed to perform data specialization (Section 3). Finally, we assess the benefits and limitations of program and data specialization based on experimental data collected by specializing a variety of programs exposing various features (Section 4).

2 Concepts of Program and Data Specialization

In this section, the basic concepts of both program and data specialization are presented. The limitations of each strategy are identified and illustrated by an example. Finally, the combination of program and data specialization is introduced.

2.1 Program Specialization

The partial evaluation community has mainly been focusing on specialization of programs. That is, given some inputs of a program, partial evaluation generates a residual program which encodes the result of the early computations which depend on the known inputs. Although program specialization has successfully been used for a variety of applications (*e.g.,* operating systems [10, 11], scientific programs [8, 12], and compiler generation [2, 6]), it has shown some limitations. One of the most fundamental limitations is code explosion which occurs when the size of the specialization problem is large. Let us illustrate this limitation using the procedure displayed on the left-hand side of Figure 1. In this example, stat is considered static, whereas dyn and d are dynamic. Static constructs are printed in boldface. Assuming the specialization process unrolls the loop, variable i becomes static and thus the gi procedures (*i.e.,* g1, g2 and g3) can be fully evaluated. Even if the gi procedures correspond to non-expensive computations, program specialization still optimizes procedure f in that it simplifies its control flow: the loop and one of the conditionals are eliminated. A possible specialization of procedure f is presented on the right-hand side of Figure 1.

However, beyond some number of iterations, the unrolling of a loop, and the computations it enables, do not pay for the size of the resulting specialized program; this number depends on the processor features. In fact, as will be shown later, the specialized program can even get slower than the unspecialized program. The larger the size of the residual loop body, the earlier this phenomenon happens.

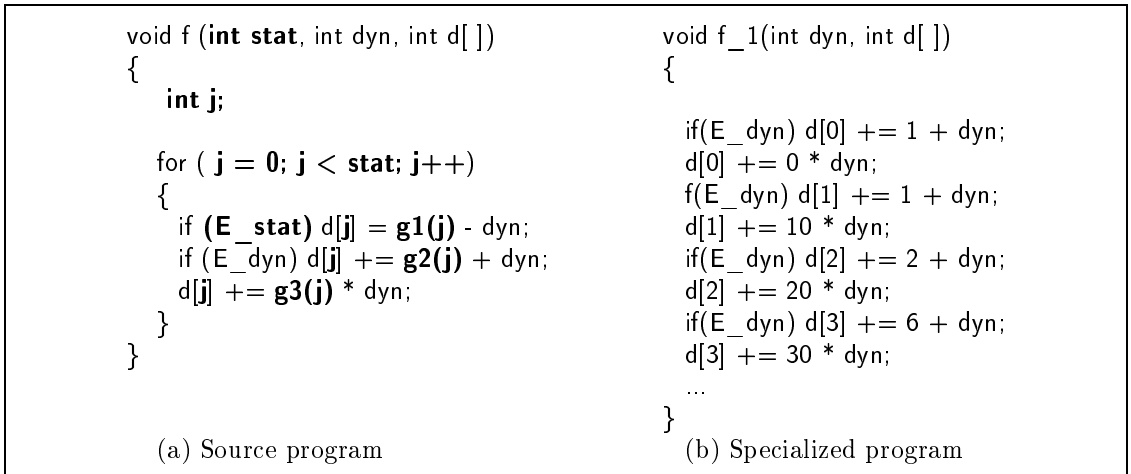

Figure 1: Program specialization

For domains like graphics and scientific computing, some applications are beyond the reach of program specialization because the specialization opportunities rely on very large data or iteration bounds which would cause code explosion if loops traversing these data were unrolled. In this situation, data specialization may apply.

2.2 Data Specialization

In late eighties, an alternative to program specialization, called data specialization, was introduced by Barzdins and Bulyonkov [1] and further explored by Malmkjær [9]. Later, Knoblock and Ruf studied data specialization for a subset of C and applied it to a graphics application [7].

Data specialization is aimed at encoding the results of early computations in data structures, not in the residual program. The execution of a program is divided into two stages: a loader first executes the early computations and saves their result in a cache. Then, a reader performs the remaining computations using the result of the early computations contained in the cache. Let us illustrate this process by an example displayed in Figure 2. On the left-hand side of this figure, a procedure f is repeatedly invoked in a loop with a first argument (c) which does not vary (and is thus considered early); its second argument, the loop index (k) varies at each iteration. Procedure f is also passed a different vector at each iteration, which is assumed to be late. Because this procedure is called repeatedly with the same first argument, data specialization can be used to perform the computations which depend on it. In this context, many computations can be performed, namely the loop test, E_stat and the invocation of the gi procedures. Of course, caching an expression assumes that its execution cost exceeds the cost of a cache reference. Measurements have shown that caching expressions which are too simple (*e.g.* a variable occurrence or simple comparisons) actually cause the resulting program to slow down.

In our example, let us assume that, like the loop test, the cost of expression E_stat is not expensive enough to be cached. If, however, the gi procedures are assumed to consist of expensive computations their invocations need to be examined as potential candidate for caching. Since the first conditional test E_stat is early, it can be put in the loader so that whenever it evaluates to true the invocation of procedure g1 can be cached; similarly, in the reader, the cache is looked up only if the conditional test evaluates to true. However, the invocation of procedure g2 cannot be cached according to Knoblock and Ruf's strategy, since it is under dynamic control and thus caching its result would amount to performing speculative evaluation [7]. Finally, the invocation of procedure g3 needs to be cached since it is unconditionally executed and its argument is early. The resulting loader and reader for procedure f are presented on the right-hand side of Figure 2, as well as their invocations.

```
extern int w[N][M];

...

for (k = 0; k < MAX; k++)
   f (c, k, w[k]);

...

void f (int stat, int dyn, int d[ ])
{
   int j;

   for (j = 0; j < stat; j++)
   {
     if (E_stat) d[j] = g1(j) - dyn;
     if (E_dyn) d[j] += g2(j) + dyn;
     d[j] += g3(j) * dyn;
   }
}
```

(a) Source program

```
extern int w[N][M];
struct data_cache { int val1;
                    int val3;} cache[MAX];
f_load (c, cache);
for (k = 0; k < MAX; k++)
  f_read (c, k, w[k], cache);
...
void f_load (stat,cache[ ])
      int stat;
      struct data_cache cache[ ];
{
  int j;
  for (j = 0; j < stat; j++) {
    if (E_stat) cache[j].val1 = g1(j);
    cache[j].val3 = g3(j);
  }
}
void f_read (stat, dyn, d[ ], cache[ ])
      int stat, dyn, d[ ];
      struct data_cache cache[ ];
{
  int j;
  for (j = 0; j < stat; j++) {
    if (E_stat) d[j] = cache[j].val1 - dyn;
    if (E_dyn) d[j] += s2(j) + dyn;
    d[j] += cache[j].val3 * dyn;
  }
}
```

(b) Specialized program

Figure 2: Data specialization

To study the limitations of data specialization consider a program where computations to be cached are not expensive enough to amortize the cost of memory reference. In our example, assume the gi procedures correspond to such computations. Then, only the control flow of procedure f remains a target for specialization.

2.3 Combining Program and Data Specialization

We have shown the benefits and limitations of both program and data specialization. The main parameters to determine which strategy fits the specialization opportunities are the cost of the early computations and the size of the specialization problem. Obviously, within the same program (or even a procedure), some fragments may require program specialization and others data specialization. As a simple example consider a procedure which consists of two nested loops. The innermost loop may require few iterations and thus allow program specialization to be applied. Whereas, the outermost loop may iterate over a vector whose size is very large; this may prevent program specialization from being applied, but not data specialization from exploiting some opportunities.

Concretely performing both program and data specialization can be done in a simple way. One approach consists of doing data specialization first, and then applying the program specializer on either the loader or the reader, or both. The idea is that code explosion may not be an issue in one of these components; as a result, program specialization can further optimize the loader or

the reader by simplifying its control flow or performing speculative specialization. For example, a reader may consist of a loop whose body is small; this situation may thus allow the loop to be unrolled without causing the residual program to be too large. Applying a program specializer to both the reader and the loader may be possible if the fragments of the program, which may cause code explosion, are made dynamic.

Alternatively, program specialization can be performed prior to data specialization. This combination requires program specialization to be applied selectively so that only fragments which do not cause code explosion are specialized. Then, the other fragments offering specialization opportunities can be processed by data specialization.

As is shown in Section 4, in practice combining both program and data specialization allows better performance than pure data specialization and prevents the performance gain from dropping as quickly as in the case of program specialization as the problem size increases.

3 Integrating Program and Data Specialization

We now present how Tempo is extended to perform data specialization. To do so let us briefly describe its features which are relevant to both data specialization and the experiments presented in the next section.

3.1 Tempo

Tempo is an off-line program specializer for C programs. As such, specialization is preceded by a preprocessing phase. This phase is aimed at computing information to guide the specialization process. The main analyses of Tempo's preprocessing phase are an alias analysis, a side-effect analyses, a binding-time analysis and an action analysis. The first two analyses are needed because of the imperative nature of the C language, whereas the binding-time analysis is typical of any off-line specializer. The action analysis is more unusual: it computes the specialization actions (*i.e.,* the program transformations) to be performed by the specialization phase.

The output of the preprocessing phase is a program annotated with specialization actions. Given some specialization values, this annotated program can be used by the specialization phase to produce a residual program at compile time, as is traditionally done by partial evaluators. In addition, Tempo can specialize a program at run time. Tempo's run-time specializer is based on templates which are efficiently compiled by standard C compilers [3, 12].

Tempo has been successfully used for a variety of applications ranging from operating systems [10, 11] to scientific programs [8, 12].

3.2 Extending Tempo with Data Specialization

Tempo includes a binding-time analysis which propagates binding times forward and backward. The forward analysis aims at determining the static computations; it propagates binding times from the definitions to the uses of variables. The backward analysis performs the same propagation in the opposite direction; when uses of a variable are both static and dynamic, its definition is annotated *static&dynamic*. This annotation indicates that the definition should be evaluated both at specialization time and run time. This process, introduced by Hornof *et al.*, allows a binding-time analysis to be more accurate; such an analysis is said to be *use sensitive* [5]. When a definition is static&dynamic and occurs in a control construct (*e.g.,* `while`), this control construct becomes static&dynamic as well. The specialized program is the code where constructs and expressions annoted static are evaluated at specialization time and its result are introduced in the residual code and where constructs and expressions annoted dynamic or static&dynamic are rebuilt in the residual code.

To perform data specialization an analysis is inserted between the forward analysis and backward analysis. In essence, this new phase identifies the *frontier terms*, that is, static terms occurring in a dynamic (or static&dynamic) context. If the cost of the frontier term is below

a given threshold (defined as a parameter of the data specializer), it is forced to dynamic (or static&dynamic).

Furthermore, because data specialization does not perform speculative evaluation, static computations which are under dynamic control are made dynamic.

Once these adjustments are done, the backward phase of the binding-time analysis then determines the final binding times of the program. Later in the process, the static computations are included in the loader and the dynamic computations in the reader; the frontier terms are cached.

The rest of our data specializer is the same as Knoblock and Ruf's.

4 Performance Evaluation

In this section, we compare the performance obtained by applying different specialization strategies on a set of programs. This set includes several scientific programs and a system program.

4.1 Overview

Machine and Compiler. The measurements presented in this paper were obtained using a Sun UltraSPARC 1 Model 170 with 448 mega bytes of main memory, running Sun-OS version 5.5.1. Times were measured using the Unix system call `getrusage` and include both "user" and "system" times.

Figure 3 displays the speedups and the size increases of compiled code obtained for different specialization strategies. For each benchmark, we give the program invariant used for specialization and an approximation of its time complexity. The code sources are included in the appendices. All the programs were compiled with `gcc -O2`. Higher degrees of optimization did not make a difference for the programs used in this experiment.

Specialization strategies. We evaluate the performance of five different specialization methods. The speedup is the ratio between the execution times of the specialized program and the original one. The size increases is the ratio between the size of the specialized program and the original one. The data displayed in Figure 3 correspond to the behavior of the following specialization strategies:

- `PS-CT`: the program is program specialized at compile time.
- `PS-RT`: the program is program specialized at run time.
- `DS`: the program is data specialized.
- `DS + PS-CT`: the program is data specialized and program specialized at compile time. The loops which manipulate the cache (for data specialization) are kept dynamic to avoid code explosion.
- `DS + PS-RT`: the program is data specialized and program specialized at run time. As in the previous strategy, the loops which manipulate the cache are kept dynamic to avoid code explosion.

Source programs. We consider a variety of source programs: a one-dimensional fast Fourier transformation (FFT), a Chebyshev approximation, a Romberg integration, a Smirnov integration, a cubic spline interpolation and a Berkeley packet filter (BPF). Given the specialization strategies available, these programs can be classified as follows.

Control flow intensive. A program which mainly exposes control flow computations; data flow computations are inexpensive. In this case, program specialization can improve performance whereas data specialization does not because there is no expensive calculations to cache.

Data flow intensive. A program which is only based on expensive data flow computations. As a result, program specialization at compile time as well as data specialization can improve the performance of such program.

Control and data flow intensive. A program which contains both control flow computations and expensive data flow computations. Such program is a good candidate for program specialization at compile time when applied to small values, and well-suited for data specialization when applied to large values.

We now analyze the performance of five specialization methods in turn on the benchmark programs.

4.2 Results

Data specialization can be executed at compile time or at run time. At run time, the loader of the cache is executed before the execution of the specialized program, while at compile time, the cache is constructed before the compilation. The cache is then used by the specialized program during the execution. For all programs, data specialization yields a greater speedup than program specialization at run time. The combination of these two specialization strategies does not make a better result.

In this section, we characterize different opportunities of specialization to illustrate our method in the three categories of program.

4.2.1 Program Specialization

We analyze two programs where performance is better with program specialization: the Berkeley packet filter (BPF), which interprets a packet with respect to an interpreter program, and the cubic spline interpolation, which approximates a function using a third degree polynomial equation.

Characteristics: For the BPF, the program consists exclusively of conditionals whose tests and branches contain inexpensive expressions. For the cubic spline interpolation, the program consists of small loops whose small body can be evaluated in part. Concretely, a program which mainly depends on the control flow graph and whose leaves contain few calculations but partially reducible, is a good candidate for program specialization. By program specialization, the control flow graph is reduced and some calculations are eliminated. Since there is no static calculation expensive enough to be efficiently cached by data specialization, the specialized program is mostly the same as the original one. For this kind of programs, only program specialization gives significant improvements: it reduces the control flow graph and it produces a small specialized program.

Applications: The BPF (Appendix F) is specialized with respect to a program (of size n). It mainly consists of the conditionals; its time complexity is linear in the size of the program and it does not contain expensive data computations. As the program does not contain any loop, the size of the specialized program is mostly the same as the original one. In Figure 3-F, program specialization at compile time and at run time yields a good speedup, whereas data specialization only improves performance marginally. The combination of program and data specialization does not improve the performance further.

The cubic spline interpolation (Appendix E) is specialized with respect to the number of points (n) and their x-coordinates. It contains three singly nested loops; its time complexity is $O(n)$. In the first two loops, more than half of the computations of each body can be completely evaluated or cached by specialization, including real multiplications and divisions. Nevertheless, there is no expensive calculation to cache, and data specialization does not improve performance significantly. The unrolled loop does not really increase the code size because of the small complexity of the program and the small body of the loop. As a consequence, for each number of points n, the

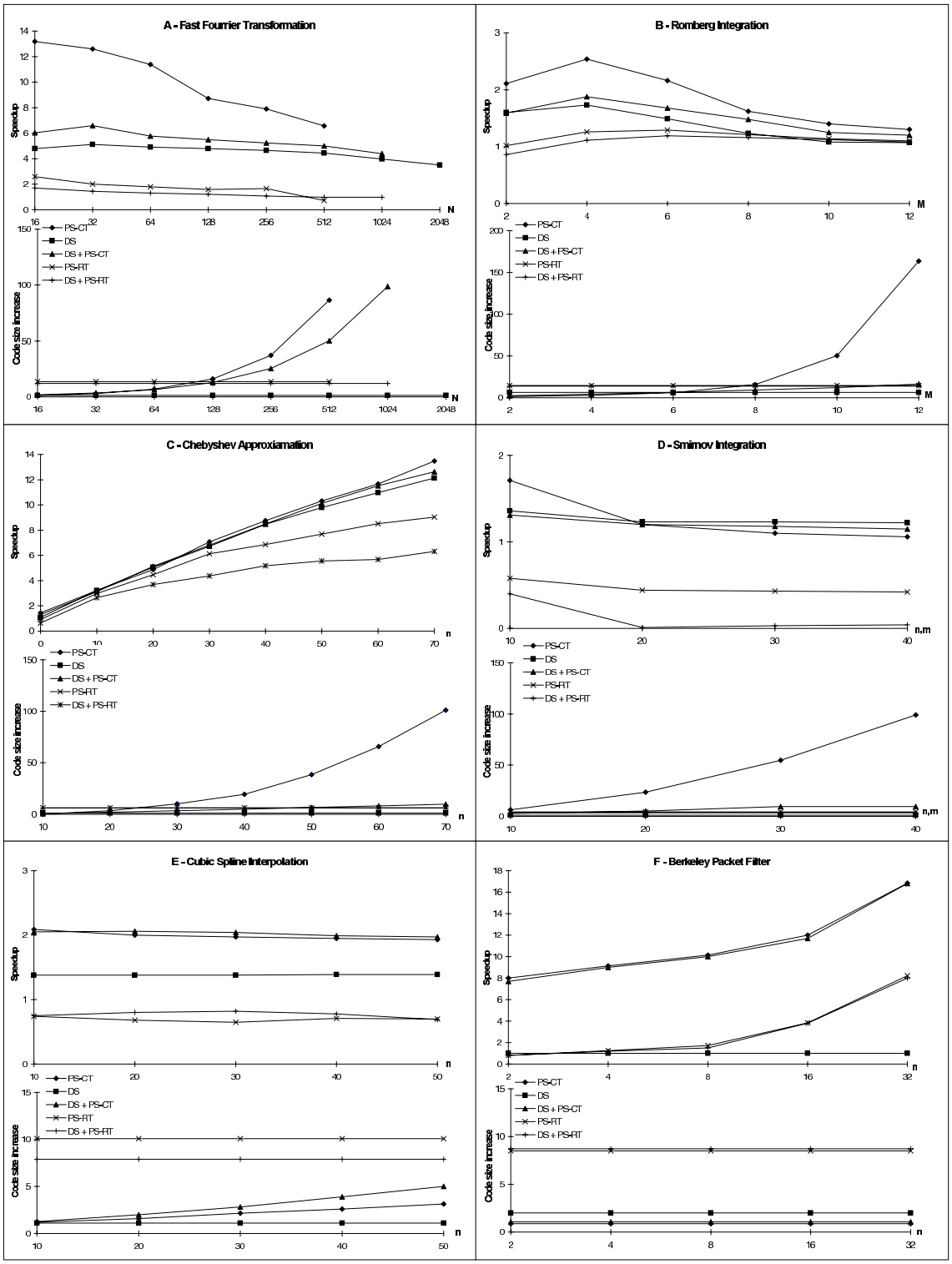

Figure 3: Program, data and combined specializations

speedup of each specialization barely changes. In Figure 3-E, program specialization at compile time produces a good speedup, whereas program specialization at run time does not improve performance. Data specialization obtains a minor speedup because the cached calculations are not expensive.

4.2.2 Program Specialization or Data Specialization

We now analyze two programs where performance is identical to program specialization or data specialization: the polynomial Chebyshev, which approximates a continuous function in a known interval, and the Smirnov integration, which approximates the integral of a function on an interval using estimations.

Characteristics: These two programs only contain loops and expensive calculations in doubly nested loops. As for the cubic spline interpolation (Section 4.2.1), more than half of the computations of each body loop can be completely evaluated or cached by specialization. In contrast with cubic spline interpolation, the static calculations in Chebyshev and Smirnov are very expensive and allow data specialization to yield major improvements. For the combined specialization, data specialization is applied to the innermost loop and program specialization is applied to the rest of the program. For this kind of programs, program and data specialization both give significant improvements. However, for the same speedup, the code size of the program produced by program specialization is a hundred times larger than the specialized program using data specialization.

Applications: The Chebyshev approximation (Appendix C) is specialized with respect to the degree (n) of the generated polynomial. This program contains two calls to the trigonometric function *cos*: one of them in a singly nested loop and the other call in a doubly nested loop. Since this program mainly consists of data flow computations, program specialization and data specialization obtain similar speedups (see Figure 3-C).

The Smirnov integration (Appendix D) is specialized with respect to the number of iterations (n, m). The program contains a call to the function *fabs* which returns the absolute value of its parameter. This function is contained in a doubly nested loop and the time complexity of this program is $O(m^n)$. As in the case of Chebyshev, program and data specialization produce similar speedups (see Figure 3-D).

4.2.3 Combining Program Specialization and Data Specialization

Finally, we analyze two programs where performance improves using program specialization when values are small, and data specialization when values are large: the FFT and the Romberg integration. The FFT converts data from the time domain to a frequency domain. The Romberg integration approximates the integral of a function on an interval using estimations.

Characteristics: These two programs contain several loops and expensive data flow computations in doubly nested loops; however more than half of the computations of each loop body cannot be evaluated. Beyond some number of iterations, when the program specialization unrolls these loops, it increases the code size of the specialized program and then degrades performance. The specialized program becomes slower because of its code size. Furthermore, beyond some problem size, the specialization process cannot produce the program because of its size. In contrast, data specialization only caches the expensive calculations, does not unroll loops, and improves performance. The result is that the code size of the program produced by program specialization is a hundred times larger than the specialized program using data specialization, for a speedup gain of 20%. The combined specialization delays the occurrence of code explosion. Data specialization is applied to the innermost loop, which contains the cache computations, and program specialization is applied to the rest of the program.

Applications: The FFT (Appendix A) is specialized with respect to the number of data points (N). It contains ten loops with several degrees of nesting. One of these loops, with complexity $O(N^2)$, contains four calls to trigonometric functions, which can be evaluated by program specialization or cached by data specialization. Due to the elimination of these expensive library calls, program specialization and data specialization produce significant speedups (see Figure 3-A). However, in the case of program specialization, code unrolling degrade performance. In contrast, data specialization produces a stable speedup regardless of the number of data points. When N is smaller than 512, data specialization does not obtain a better result in comparison to program specialization. However, when N is greater than 512, program specialization becomes impossible to apply because of the specialization time and the size of the residual code. In this situation, data specialization still gives better performance than the unspecialized program. Because this program also contains some conditionals, combined specialization, where the innermost loop is not unrolled, improves performance better than data specialization alone.

The Romberg integration (Appendix B) is specialized with respect to the number of iterations (M) used in the approximation. The Romberg integration contains two calls to the costly function *intpow*. It is called twice: once in a singly nested loop and another time in a doubly nested loop. Because both specialization strategies eliminate these expensive library calls, the speedup is consequently good. As for FFT, loop unrolling causes the program specialization speedup to decrease, whereas the data specialization speedup still remains the same, even when M increases (Figure 3-B).

5 Conclusion

We have integrated program and data specialization in a specializer named Tempo. Importantly, data specialization can re-use most of the phases of an off-line program specializer.

Because Tempo now offers both program and data specialization, we have experimentally compared both strategies and their combination. This evaluation shows that, on the one hand program specialization typically gives better speed-up than data specialization for small problem size. However, as the problem size increases, the residual program may become very large and often slower than the unspecialized program. On the other hand, data specialization can handle large problem size without much performance degradation. This strategy can, however, be ineffective if the program to be specialized mainly consists of control flow computations. The combination of both program and data specialization is promising: it can produce a residual program more efficient than with data specialization alone, without dropping in performance as dramatically as program specialization, as the problem size increases.

Acknowledgments

We thank Renaud Marlet for thoughtful comments on earlier versions of this paper, as well as the Compose group for stimulating discussions.

A substantial amount of the research reported in this paper builds on work done by the authors with Scott Thibault on Berkeley packet filter and Julia Lawall on Fast Fourier Transformation.

References

[1] G. J. Barzdins and M. A. Bulyonkov. Mixed computation and translation: Linearisation and decomposition of compilers. Preprint 791 from Computing Centre of Sibirian division of USSR Academy of Sciences, p.32, Novosibirsk, 1988.

[2] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 493–501, Charleston, SC, USA, January 1993. ACM Press.

[3] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 145–156, St. Petersburg Beach, FL, USA, January 1996. ACM Press.

[4] B. Guenter, T.B. Knoblock, and E. Ruf. Specializing shaders. In *Computer Graphics Proceedings*, Annual Conference Series, pages 343–350. ACM Press, 1995.

[5] L. Hornof, J. Noyé, and C. Consel. Effective specialization of realistic programs via use sensitivity. In P. Van Hentenryck, editor, *Proceedings of the Fourth International Symposium on Static Analysis, SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 293–314, Paris, France, September 1997. Springer-Verlag.

[6] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice-Hall, June 1993.

[7] T.B. Knoblock and E. Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 215–225, Philadelphia, PA, May 1996. ACM SIGPLAN Notices, 31(5). Also TR MSR-TR-96-04, Microsoft Research, February 1996.

[8] J.L. Lawall. Faster Fourier transforms via automatic program specialization. Publication interne 1192, IRISA, Rennes, France, May 1998.

[9] K. Malmkjaer. Program and data specialization: Principles, applications, and self-application. Master's thesis, DIKU University of Copenhagen, August 1989.

[10] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[11] G. Muller, E.N. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 116–125, Amsterdam, The Netherlands, June 1997. ACM Press.

[12] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization : Implementation and experimental study. In *International Conference on Computer Languages*, pages 132–142, Chicago, IL, May 1998. IEEE Computer Society Press. Also available as IRISA report PI-1065.

A Fast Fourrier Transformation

```
#define PI  3.14159265358979323846

int fft(np, x, y)
   int np ;
   double x[2];
   double y[2];
{
   double *px,*py;
   int i,j,k,m,n;
   int i0,i1,i2,i3;
   int is,id;
   int n1,n2,n4;
   double  a,e,a3;
   double  cc1,ss1,cc3,ss3;
   double  r1,r2;
   double  s1,s2,s3;
   double  xt;

   px = x - 1;
   py = y - 1;
   i = 2;
   m = 1;
   while (i < np)
   {
      i = i+i;
      m = m+1;
   };
   n = i;
   if (n != np)
   {
      for (i = np+1; i <= n; i++)
      {
         *(px + i) = 0.0F;
         *(py + i) = 0.0F;
      };
      printf("\nuse %d point fft",n);
   }

   n2 = n+n;
   for (k = 1;  k <= m-1; k++ )
   {
      n2 = n2 / 2;
      n4 = n2 / 4;
      e = 2.0F * (float)PI / n2;
      a = 0.0F;
      for (j = 1; j<= n4 ; j++)
      {
         a3 = 3.0F*a;
         cc1 = cos(a);
         ss1 = sin(a);
         cc3 = cos(a3);
         ss3 = sin(a3);
         a = j*e;
         is = j;
         id = 2*n2;
         while ( is < n )
         {
            for (i0 = is; i0 <= n-1;)
            {
               i1 = i0 + n4;
               i2 = i1 + n4;
               i3 = i2 + n4;

               r1 = *(px+i0) - *(px+i2);
               *(px+i0) = *(px+i0) + *(px+i2);
               r2 = *(px+i1) - *(px+i3);
               *(px+i1) = *(px+i1) + *(px+i3);
               s1 = *(py+i0) - *(py+i2);
               *(py+i0) = *(py+i0) + *(py+i2);
               s2 = *(py+i1) - *(py+i3);
               *(py+i1) = *(py+i1) + *(py+i3);

               s3 = r1 - s2;
               r1 = r1 + s2;
               s2 = r2 - s1;
               r2 = r2 + s1;

               *(px+i2) = r1*cc1 - s2*ss1;
               *(py+i2) = -s2*cc1 - r1*ss1;
               *(px+i3) = s3*cc3 + r2*ss3;
               *(py+i3) = r2*cc3 - s3*ss3;

               i0 = i0 + id;
            }
            is = 2*id - n2 + j;
            id = 4*id;
         }
      }
   }

/*
-------Last stage, length=2 butterfly---------
*/
   is = 1;
   id = 4;
   while ( is < n)
   {
      for (i0 = is; i0 <= n;)
      {
         i1 = i0 + 1;

         r1 = *(px+i0);
         *(px+i0) = r1 + *(px+i1);
         *(px+i1) = r1 - *(px+i1);

         r1 = *(py+i0);
         *(py+i0) = r1 + *(py+i1);
         *(py+i1) = r1 - *(py+i1);

         i0 = i0 + id;
      }
      is = 2*id - 1;
      id = 4 * id;
   }

/*
----------Bit reverse counter------------
*/
   j = 1;
   n1 = n - 1;
   for (i = 1; i <= n1; i++)
   {
      if (i < j)
      {
         xt = *(px+j);
         *(px+j) = *(px+i);
         *(px+i) = xt;

         xt = *(py+j);
         *(py+j) = *(py+i);
         *(py+i) = xt;
      }
      k = n / 2;
      while (k < j)
      {
         j = j - k;
         k = k / 2;
      }
      j = j + k;
   }
   return(n);

}
```

B Romberg Integration

```
void romberg(float (*r)[50], float a, float b, int M)
{
  int n, m, i, max;
  float h, s;

  h = b - a;
  r[0][0] = (f(a) + f(b)) * h / 2.0;

  for (n = 1; n <= M; n++) {
    h = h / 2.0;
    s = 0.0;

    max = int_pow(2, n - 1);
    for (i = 1; i <= max; i++) {
      s = s + f(a + (float)(2.0 * i - 1) * h);
    }

    r[n][0] = r[n-1][0]/2.0 + h * s;

    for(m = 1; m <= n; m++) {
      r[n][m] = r[n][m-1] + (float)(1.0/(int_pow(4,m)-1))
                * (r[n][m-1] - r[n-1][m-1]);
    }
  }
}

int int_pow(int base, int expon)
{
  int accum = 1;

  while (expon > 0) {
    accum *= base;
    expon--;
  }

  return(accum);
}
```

C Chebyshev Approximation

```
#define MAX1 100
#define PI  3.14159265358979323846

void cheb(float c[MAX1], int n, float xa, float xb)
{
  int k, j;
  float xm, xp, sm;
  float f[MAX1];

  xp = (xb + xa) / 2;
  xm = (xb - xa) / 2;

  for(k = 1; k <= n; k++) {
    f[k] = func(xp + xm * cos(PI * (k - 0.5) / n));
  }

  for(j = 0; j <= n-1; j++) {
    sm = 0.0;
    for(k = 1; k <= n; k++) {
      sm = sm + f[k] * cos(PI * j * (k - 0.5) / n);
    }
    c[j] = (2.0 / n) * sm;
  }
  return;
}
```

D Smirnov Integration

```
double smirnov(int m, int n, double D, double *u)
  {
    double c;
    double W;
    int i;
    int j;
    double temp;

    c = (double)(m * n) * D - 1.0 ;
    for (j = 0; j <= n; j++)
      {
        u[j] = 1.0;
        if (c < (double)(m * j))
          {
            u[j] = 0.0;
          }
      }
    for (i = 1; i <= m; i++)
      {
        double *suif_tmp0;

        W = (double)i / (double)(i + n) ;
        suif_tmp0 = u;
        *suif_tmp0 = *suif_tmp0 * W;
        if (c < (double)(n * i))
          {
            *u = 0.0;
          }
        for (j = 1; j <= n; j++)
          {
            u[j] = W * u[j] + u[j - 1]+1;
            temp=(double)fabs(n * i - m * j);
            if (c < temp)
              {
                u[j] = 0.0;
              }
          }
      }
    return 1.0 - u[n];
  }
```

E Cubic Spline Interpolation

```
#define MAX 100

void csi(int n, float x[MAX], float y[MAX], float z[MAX])
{
  float h[MAX],b[MAX],u[MAX],v[MAX];

  int i;

  for (i = 0; i<= n-1; i=i+1){
    h[i] = x[i+1]-x[i];
    b[i] = (6/h[i])*(y[i+1]-y[i]);
  }

  u[1] = 2*(h[0]+h[1]);
  v[1] = b[1]-b[0];
  for (i = 2; i <= n-1; i=i+1){
    u[i] = 2*(h[i]+h[i-1])-h[i-1]*h[i-1]/u[i-1];
    v[i] = b[i]-b[i-1]-h[i-1]*v[i-1]/u[i-1];
  }
  z[n] = 0;

  i=n-1;
  if ( i>=1 )
    do {
      z[i] = (v[i]-h[i]*z[i+1])/u[i];
      i=i-1;
    } while ( i>=1 );
  z[0] = 0;
}
```

F Berkeley Packet Filter

```
u_int bpf(pc, p, wirelen, buflen)
   register struct bpf_insn *pc;
   register u_char *p;
   u_int wirelen;
   register u_int buflen;
{
   register u_int32 A, X;
   register int k;
   int32 mem[BPF_MEMWORDS];
   u_char returned=FALSE;

   if (pc == 0)
      /*
       * No filter means accept all.
       */
      return (u_int)-1;
   A = 0;
   X = 0;
   --pc;
   while (!returned) {
      ++pc;
      switch (pc->code) {

      default:
         returned=TRUE;
         abort();
         break;

      case BPF_RET|BPF_K:
         returned=TRUE;
         return (u_int)pc->k;
         break;

      case BPF_RET|BPF_A:
         returned=TRUE;
         return (u_int)A;
         break;

      case BPF_LD|BPF_W|BPF_ABS:
         k = pc->k;
         if (k + sizeof(int32) > buflen) {
            returned=TRUE;
            return 0;
         }
         A = EXTRACT_LONG(&p[k]);
         break;

      case BPF_LD|BPF_H|BPF_ABS:
         k = pc->k;
         if (k + sizeof(short) > buflen) {
            returned=TRUE;
            return 0;
         }
         A = EXTRACT_SHORT(&p[k]);
         break;

      case BPF_LD|BPF_B|BPF_ABS:
         k = pc->k;
         if (k >= buflen) {
            returned=TRUE;
            return 0;
         }
         A = p[k];
         break;

      case BPF_LD|BPF_W|BPF_LEN:
         A = wirelen;
         break;

      case BPF_LDX|BPF_W|BPF_LEN:
         X = wirelen;
         break;

      case BPF_LD|BPF_W|BPF_IND:
         k = X + pc->k;
         if (k + sizeof(int32) > buflen) {
            returned=TRUE;
            return 0;
         }
         A = EXTRACT_LONG(&p[k]);
         break;

      case BPF_LD|BPF_H|BPF_IND:
         k = X + pc->k;
         if (k + sizeof(short) > buflen) {
            returned=TRUE;
            return 0;
         }
         A = EXTRACT_SHORT(&p[k]);
         break;

      case BPF_LD|BPF_B|BPF_IND:
         k = X + pc->k;
         if (k >= buflen) {
            returned=TRUE;
            return 0;
         }
         A = p[k];
         break;

      case BPF_LDX|BPF_MSH|BPF_B:
         k = pc->k;
         if (k >= buflen) {
            returned=TRUE;
            return 0;
         }
         X = (p[pc->k] & 0xf) << 2;
         break;

      case BPF_LD|BPF_IMM:
         A = pc->k;
         break;

      case BPF_LDX|BPF_IMM:
         X = pc->k;
         break;

      case BPF_LD|BPF_MEM:
         A = mem[pc->k];
         break;

      case BPF_LDX|BPF_MEM:
         X = mem[pc->k];
         break;

      case BPF_ST:
         mem[pc->k] = A;
         break;

      case BPF_STX:
         mem[pc->k] = X;
         break;

      case BPF_JMP|BPF_JA:
         pc += pc->k;
         break;

      case BPF_JMP|BPF_JGT|BPF_K:
         pc += (A > pc->k) ? pc->jt : pc->jf;
         break;

      case BPF_JMP|BPF_JGE|BPF_K:
         pc += (A >= pc->k) ? pc->jt : pc->jf;
         break;

      case BPF_JMP|BPF_JEQ|BPF_K:
         pc += (A == pc->k) ? pc->jt : pc->jf;
         break;

      case BPF_JMP|BPF_JSET|BPF_K:
         pc += (A & pc->k) ? pc->jt : pc->jf;
         break;

      case BPF_JMP|BPF_JGT|BPF_X:
         pc += (A > X) ? pc->jt : pc->jf;
         break;

      case BPF_JMP|BPF_JGE|BPF_X:
```

```
   pc += (A >= X) ? pc->jt : pc->jf;
   break;

case BPF_JMP|BPF_JEQ|BPF_X:
   pc += (A == X) ? pc->jt : pc->jf;
   break;

case BPF_JMP|BPF_JSET|BPF_X:
   pc += (A & X) ? pc->jt : pc->jf;
   break;

case BPF_ALU|BPF_ADD|BPF_X:
   A += X;
   break;

case BPF_ALU|BPF_SUB|BPF_X:
   A -= X;
   break;

case BPF_ALU|BPF_MUL|BPF_X:
   A *= X;
   break;

case BPF_ALU|BPF_DIV|BPF_X:
        if (X == 0) {
      returned=TRUE;
      return 0;
   }
   A /= X;
   break;

case BPF_ALU|BPF_AND|BPF_X:
   A &= X;
   break;

case BPF_ALU|BPF_OR|BPF_X:
   A |= X;
   break;

case BPF_ALU|BPF_LSH|BPF_X:
   A <<= X;
   break;

case BPF_ALU|BPF_RSH|BPF_X:
   A >>= X;
   break;

case BPF_ALU|BPF_ADD|BPF_K:
   A += pc->k;
   break;

case BPF_ALU|BPF_SUB|BPF_K:
   A -= pc->k;
   break;

case BPF_ALU|BPF_MUL|BPF_K:
   A *= pc->k;
   break;

case BPF_ALU|BPF_DIV|BPF_K:
   A /= pc->k;
   break;

case BPF_ALU|BPF_AND|BPF_K:
   A &= pc->k;
   break;

case BPF_ALU|BPF_OR|BPF_K:
   A |= pc->k;
   break;

case BPF_ALU|BPF_LSH|BPF_K:
   A <<= pc->k;
   break;

case BPF_ALU|BPF_RSH|BPF_K:
   A >>= pc->k;
   break;

case BPF_ALU|BPF_NEG:
   A = -A;
   break;

case BPF_MISC|BPF_TAX:
   X = A;
   break;

case BPF_MISC|BPF_TXA:
   A = X;
   break;
   }
  }
  return 0;
}
```