Higher-Order and Symbolic Computation, , 1–20 (1999)
© 1999 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Combining Program and Data Specialization

SANDRINE CHIROKOFF, CHARLES CONSEL AND RENAUD MARLET

{sandrine.chirokoff,charles.consel,renaud.marlet}@irisa.fr
Compose project, IRISA / INRIA - Université de Rennes 1, Campus Universitaire de Beaulieu, 35042 Rennes cedex, France, http://www.irisa.fr/compose

Abstract.
Program and data specialization have always been studied separately, although they are both aimed at processing early computations. Program specialization encodes the result of early computations into a new program; while data specialization encodes the result of early computations into 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 single 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.

Keywords: program transformation, partial evaluation, program specialization, data specialization, combining program and data specialization

1. Introduction

Program and data specialization both aim at performing computations which depend on values that are available early. 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 useful to calculate the final result with the 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 that 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 by the specialized program.

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. Data specialization splits the execution of a 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 specialized 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. This 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 results of early computations from the program which exploits them. 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 that are far beyond the reach of program specialization, and thus opens up new opportunities as demonstrated by Knoblock and Ruf for graphics applications [6, 11]. However, data specialization, as implemented to date, does not optimize the control flow: it is limited to performing the early computations which are expensive enough to be worth caching. 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, we present the basic concepts of both program and data specialization. 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 focused on program specialization. That is, given some inputs of a program, program specialization generates a residual program that encodes the result of the early computations which depend on the early inputs. Although program specialization has successfully been used for a variety of applications (*e.g.,* operating systems [14, 15], scientific programs [12, 16], and compiler generation [2, 10]), 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, the input `stat` is an early data, whereas the inputs `dyn` and `d` are late data. `E_stat` (respectively `E_dyn`) stand for an early (respectively late) expression. Assuming the specialization process unrolls the loop, variable `j` becomes available and thus the `g`*i* procedures (*i.e.,* `g1`, `g2` and `g3`) can be fully evaluated. Additionally, program specialization optimizes procedure `f` by simplifying 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.

Beyond some number of iterations however, the unrolling of a loop, and the computations it enables, do not pay for the code size of the resulting specialized program; this number depends on behavior of the processor used to execute the specialized program. In fact, the specialized program can even be slower than the unspecialized program. The larger the size of the residual loop body, the earlier this phenomenon happens.

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 that would cause code explosion if loops traversing these data were unrolled. Certainly, code explosion can cause code size problems resulting in an increase of instruction cache misses and poor performance of the specialized program. In this situation, data specialization may apply.

```
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;
  }
}


/****** main ******/
extern int w[N][M];
...
for (k=0; k<MAX; k++)
  f(c,k,w[k]);
```

(a) Source program

```
void f_100 (int dyn,int d[])
{
  if (E_dyn) d[0]+=1+dyn;
  d[0]+=0*dyn;
  d[1]=3-dyn;
  if (E_dyn) d[1]+=1+dyn;
  d[1]+=10*dyn;
  if (E_dyn) d[2]+=2+dyn;
  d[2]+=20*dyn;
  d[3]=12-dyn;
  if (E_dyn) d[3]+=6+dyn;
  ...
}
/****** main ******/
extern int w[N][M];
...
for (k=0; k<MAX; k++)
  f_100(k,w[k]);
```

(b) Specialized program

Figure 1. Program specialization: early constructs are shown in bold. The procedure `f` is specialized for `stat=100`, causing the loop to be unrolled 100 times.

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 [13]. Later, Knoblock and Ruf studied data specialization for a subset of C and applied it to a graphics application [11].

Data specialization aims 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. First, a *loader* executes the early computations and saves their result in a data structure, the *cache*. This stage specializes the cache given the available inputs. Then, a *reader* performs the remaining computations using the results shared in the specialized 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

```
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;
  }
}

/****** main ******/
extern int w[N][M];

...

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

(a) Source program

```
struct D_cache {int V1[MAX];
                int V2[MAX];};

void f_loader (int stat,
               struct D_cache *C)
{int j;
  for (j=0; j<stat; j++){
    if (E_stat) C->V1[j]=g1(j);
    C->V2[j]=g3(j);
} }

void f_reader(int stat,int dyn,int d[]
             ,struct D_cache *C)
{int j;
  for (j=0; j<stat; j++){
    if (E_stat) d[j]=C->V1[j]-dyn;
    if (E_dyn)  d[j]+=g2(j)+dyn;
    d[j]+=C->V2[j]*dyn;
} }

/****** main ******/
extern int w[N][M];
struct D_cache *cache;
...
f_loader(c,cache);
for (k=0; k<MAX; k++)
  f_reader(c,k,w[k],cache);
```

(b) Specialized program

Figure 2. Data specialization: early constructs are shown in bold and cached expressions are underlined. The procedure `f` is specialized for `stat` being available. The cache is load by `f loader` and read by `f reader`.

argument, data specialization can be used to perform the computations that depend on it. In this context, many computations can be performed, namely the loop test, `E_stat` and the invocation of the `g`i procedures. Of course, caching an expression is only beneficial if its execution cost exceeds the cost of a cache reference. Measurements have shown that caching expressions that are too simple (*e.g.* a variable occurrence or simple comparisons) actually cause the resulting program to be slower than the original program. The reason is that looking up the cache for the value of an early expression requires a memory access, whereas computing the expression at program execution time may only involve registers.

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 `g`i 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 late control and thus caching its result would amount to performing speculative evaluation [11]. Finally, the invocation of procedure `g3` can 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.

When performing program specialization, it is always advantageous to simplify early expressions. However, when performing data specialization, it is not always desirable to store the results of such expressions in the cache. If the computation performed by calling the `g`i procedures is not expensive enough to amortize the cost of memory references to the cache, then, only the control flow of procedure `f` remains a target for specialization which data specialization cannot exploit. This is a limitation of data specialization.

2.3. Combining Program and Data Specialization

We have described the benefits and limitations of both program and data specialization. The main parameters that determine which strategy fits the specialization opportunities are the cost of the early computations and the size of the specialization problem. Nevertheless, within a single program (or even a single procedure), some fragments may require program specialization and others data specialization. As a simple example, consider a procedure that consists of two singly nested loops. Suppose the body of the first loop can be almost completely evaluated, and the number of iterations is small, then program specialization can be applied. On the contrary, assume the body of the second loop is large and cannot be completely evaluated, it can cause code explosion; this may prevent program specialization from being applied, but it does not prevent data specialization from exploiting some specialization opportunities.

Concretely program and data specialization can be combined 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 evaluation. For example, a reader may contain 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 is possible if the fragments of the program that may cause code explosion are not unrolled by the specialization phase.

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

These two approaches give an identical result on the reader since they simply consist of applying data specialization on the fragments of program that may cause code explosion, and applying program specialization on all of the program without modifying the already specialized fragments. Suppose we rewrite `f` to calculate `d[]` using two singly nested loops, as shown in Figure 3. The first loop, that initializes the values of `d[]`, contains an early test that can be evaluated by program specialization. The second loop contains the rest of the calculation and can be optimized by data specialization. The resulting specialized loader and reader for the procedure `f` are presented on the right-hand side of Figure 3, together with their invocations. The first loop has been program-specialized to

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

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


/****** main ******/
extern int w[N][M];

...

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

(a) Source program

```
struct D_cache {int V1[100];};

void f_loader_100 (struct D_cache *C)
{int j;
  for (j=0; j<100; j++){
    C->V1[j]=g3(j);
} }

void f_reader_100 (int dyn,int d[],
                   struct D_cache *C)
{int j;
  d[1]=3-dyn;
  d[3]=12-dyn;
  ...
  for (j=0; j<100; j++){
    if (E_dyn)  d[j]+=g2(j)+dyn;
    d[j]+=C->V1[j]*dyn;
} }

/****** main ******/
extern int w[N][M];
struct D_cache *cache;
...
f_loader_100(cache);
for (k=0; k<MAX; k++)
  f_reader_100(k,w[k],cache);
```

(b) Specialized program

Figure 3. Program and data specialization: early constructs are shown in bold and cached expressions are underlined. The procedure `f` is specialized for `stat=100`, eliminating the first singly nested loop and its conditional, but without unrolling the second singly nested loop.

initialize `d[ ]`, and the loop and the conditional have been eliminated. The second loop was not unrolled to prevent code explosion, but the size of the cache with regards to data specialization alone and the code size with regards to program specialization alone were decreased.

As will be 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 a program specializer is extended to perform data specialization in practice. To do so we briefly describe the features of Tempo that 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 [3, 4]. As such, specialization is preceded by a preprocessing phase. This phase computes information that guides the specialization process. The main analyses of Tempo's preprocessing phase are an alias analysis, a side-effect analysis, a binding-time analysis and an action analysis [7, 8]. The first two analyses are needed because of the imperative nature of the C language, the binding-time analysis is typical of any off-line specializer. The action analysis is more unusual: it computes specialization actions (*i.e.,* program transformations) to be performed by the specialization phase.

The output of the preprocessing phase is a program annotated with specialization actions. Given specialization values, this annotated program is used by the specialization phase to produce a residual program.

Tempo has been successfully used for a variety of applications ranging from operating systems [14, 15, 17, 18] to scientific programs [7, 12, 16] and is publicly available[1].

3.2. Extending Tempo with Data Specialization

Tempo includes a binding-time analysis which propagates binding time annotations forward and backward. A *binding-time annotation* indicates whether the value of a variable is available at specialization time. An early expression is annotated with the *static* binding-time annotation and a late expression is annotated with the *dynamic* binding-time annotation. Thus an expression is considered static (respectively dynamic) when it depends on early (respectively late) data.

3.2.1. Program specialization

The forward analysis aims at determining the static computations; it propagates binding times from the definitions to the uses of variables. If the binding time of a use is static, then the value of this variable is available at specialization time.

The backward analysis performs the same propagation in the opposite direction; when uses of a variable are both static and dynamic, the corresponding definition is annotated *static&dynamic*. This annotation indicates that the definition should both be evaluated at specialization time for the static uses and introduced in the residual code for the dynamic uses. This process, introduced by Hornof *et al.* [9], allows a binding-time analysis to be accurate; such an analysis is said to be *use sensitive* . 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 formed as follows: statements and expressions annotated static or static&dynamic are evaluated at specialization time; the results of static expressions are introduced in the residual code; statements and expressions annotated dynamic or static&dynamic are rebuilt in the residual code.

3.2.2. Data specialization

Program and data specialization share the same forward analysis; it aims at determining the static computations. After the forward analysis, a new analysis phase focuses on the computations which can be cached, that is, static terms occurring in a dynamic (or static&dynamic) context. These terms are called *frontier terms*. If the cost of the frontier term is below a given threshold, it must not be cached: it is forced dynamic (or static&dynamic). The calculation of the threshold is defined as a parameter of the data specializer in order to select the frontier terms and to limit the cache size, according to the system architecture.

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

Once these adjustments are done, the backward analysis performs the propagation of uses in the opposite direction; it identifies the computations needed to calculate the cache and the computations needed to compute the result of the program using the cache. The result of the specialization is a program where statements and expressions annotated static are included in the loader for building the cache. Statements and expressions annotated dynamic are included in the reader for evaluating the result using the cache. Static&dynamic expressions are present both in the loader and in the reader.

4. Performance Evaluation

We now compare the performance obtained by applying different specialization strategies to a set of programs. This set includes several scientific programs and a systems program.

4.1. Overview

Machine and Compiler. The measurements presented in this paper were obtained using a Sun UltraSPARC 1 Model 170 with 448 megabytes of main memory and 16 kilobytes of instruction and data cache, running SunOS version 5.5.1. Times were measured using the Unix system call `getrusage` and include both “user” and “system” times. Instruction and data cache misses were measured using the program call `perf-monitor`, which uses the on-chip counters on UltraSPARC processors to gather statistics.

Figures 4, 5 and 6 display the speedups, the code size increases, and the data size increases obtained for different specialization strategies. The speedup is the ratio between the execution times (without specialization times) of the specialized program and the original one. In the following, for each benchmark, we give the program invariants used for specialization and an approximation of the time complexity of the program. Figure 7 displays the increase in instruction and data cache misses. Measurements indicated for each figures represent the ratio of increase compared to the original program.

The source code is included in the appendices. All the programs were compiled with `gcc` at optimization level `O2`. Higher degrees of optimization did not make a difference for the programs used in this experiment.

Specialization strategies. We evaluate the performance of three different specialization methods. The data displayed in Figure 4, 5 and 6 correspond to the behavior of the following specialization strategies:

- PS: the program is program-specialized.
- DS: the program is data-specialized.
- DS + PS: the program is data-specialized and program-specialized.

  Loops that manipulate the cache (for data specialization) are kept folded to avoid code explosion. Because the cached computations are sufficiently expensive, the program produced by the combined strategy performs better than a program in which the loops are kept folded but where the static computations are not cached.

Source programs. We consider a variety of source programs: a Chebyshev approximation, a Romberg integration, a cubic spline interpolation, a one-dimensional fast Fourier transformation (FFT), a Smirnov integration and the Berkeley packet filter (BPF). The first three programs come from a paper of Robert Glück [5]. Given the specialization strategies available, these programs can be classified as follows.

Control intensive: a program that mainly depends on control computations, and where data 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 intensive: a program that is only based on expensive data computations. As a result, program specialization as well as data specialization can improve the performance of such program.

Control and data intensive: a program that contains both control computations and expensive data computations. Such program is a good candidate for program specialization when applied to small values, and well-suited for data specialization when applied to large values.

We now analyze in turn the performance of the three specialization strategies on the benchmark programs.

4.2. *Results*

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

4.2.1. *Control intensive*

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

Characteristics: The BPF interpreter consists exclusively of conditionals whose tests and branches contain inexpensive expressions. The implementation of the cubic spline interpolation algorithm consists of small loops whose bodies can be evaluated in part. These programs mainly depend on control and contain few calculations. Program specialization simplifies the control flow and eliminates some calculations. Since there is no static calculation expensive enough to be efficiently cached by data specialization, the data-specialized program is mostly the same as the original one. For this kind of program, only program specialization gives significant improvements: it reduces the number of tests and it produces a small specialized program.

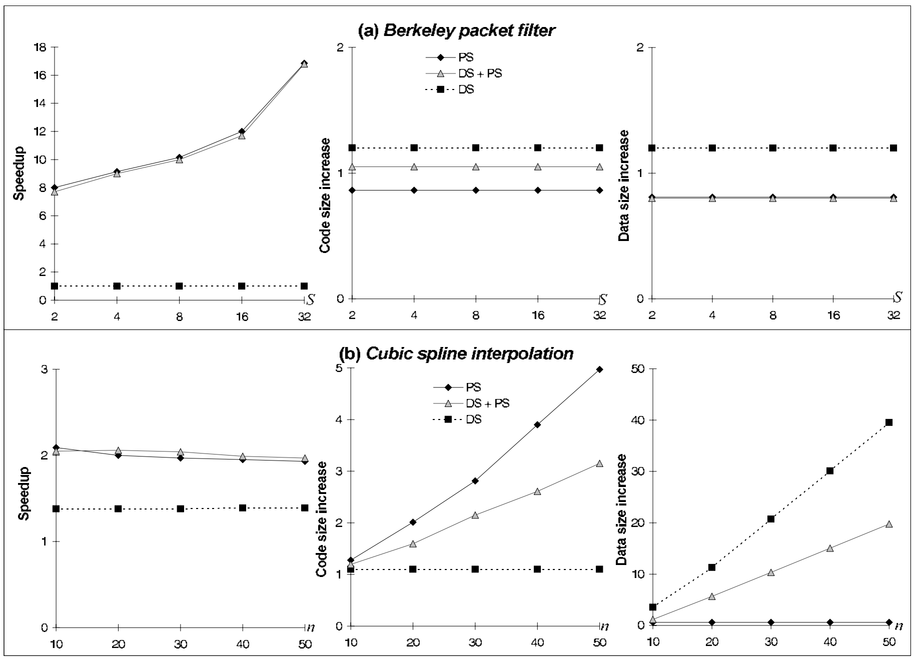

Figure 4. Control intensive : program, data and combined specialization

Applications: The BPF interpreter (Appendix A.6) is specialized with respect to a program pc, and mainly consists of conditionals. Its time complexity is linear in the size S of the interpreted program and it does not contain expensive data computations. As the BPF interpreter does not contain any loops, the size of the specialized program is mostly the same as the original one. In Figure 4 (a), program specialization yields a good speedup, whereas data specialization does not improve performance because there are no expensive data computations to be cached. So data specialization does not modify the original program. The combination of program and data specialization corresponds exactly to the residual program produced by program specialization.

The cubic spline interpolation program (Appendix A.5) is specialized with respect to the number of points (n) and the x-coordinate. 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 floating-point multiplications and divisions. Nevertheless, no individual computation is expensive enough to warrant caching, so data specialization does not improve performance significantly. The unrolled loops do not significantly increase the code because of the linear time complexity of the program and the small size of each loop body. In the same way, the data size increases remain linear. So for each number of points n, the speedup resulting from each specialization strategy is not degraded by these weak increases. In Figure 4 (b), program specialization produces a good enough speedup, 45% better than data specialization for this example. Data specialization obtains only a minor speedup because the cached calculations are not very expensive. For the combination of program and data specialization, we kept as folded the loop that contains the most expensive cached computations. We obtain a similar speedup as with program specialization.

4.2.2. Data intensive

We now analyze two programs where performances of the specialized versions are similar: the Chebyshev polynomial, 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, more than half of the computations of each loop body 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 can typically be a hundred times larger than the program specialized using data specialization.

Applications: The Chebyshev approximation (Appendix A.3) 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. Its time complexity is $O(n^2)$. Since this program mainly consists

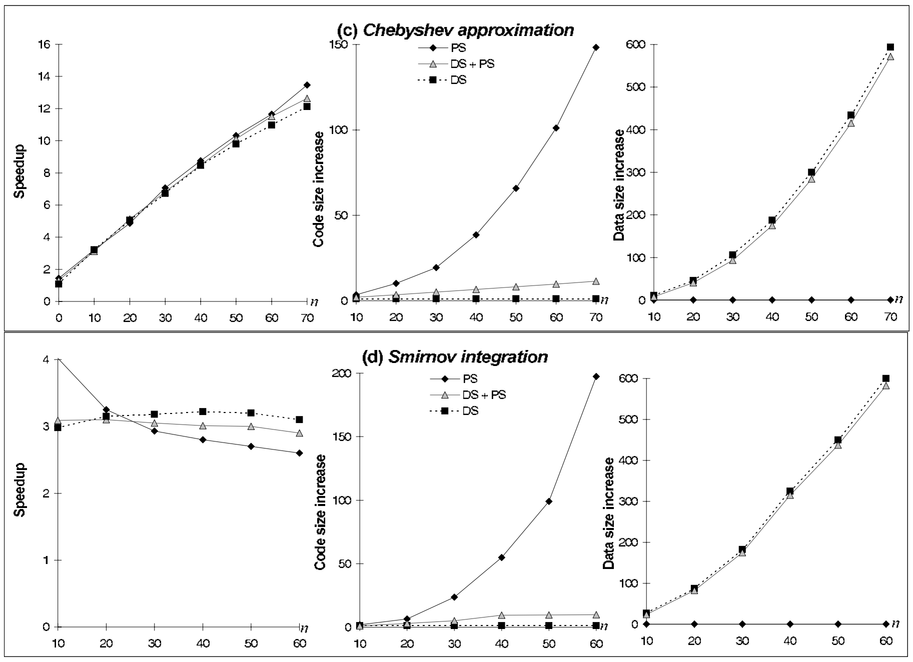

Figure 5. Data intensive : program, data and combined specialization

of data flow computations, program specialization and data specialization obtain similar speedups (see Figure 5 (c)).

The Smirnov integration (Appendix A.4) is specialized with respect to the number of iterations (n, m), and where $n = m$. 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 $O(mn)$. As in the case of the Chebyshev approximation, program and data specialization produce similar speedups (see Figure 5 (d)).

4.2.3. Control and data intensive

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 at specialization time. Beyond some number of iterations, when program specialization unrolls these loops, the increase of the code size in the

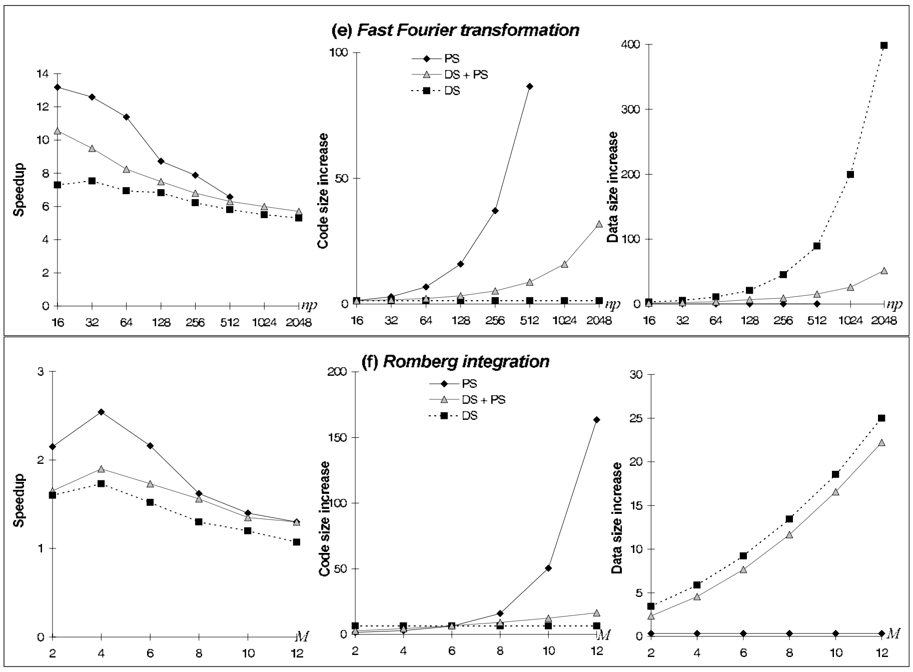

Figure 6. Control and data intensive : program, data and combined specialization

specialized program degrades performance. For an even larger number of iterations, the specialization phase of Tempo cannot even produce the specialized program because of excessive memory consumption. In contrast, data specialization caches only the expensive calculations, does not unroll loops, and improves performance. When the number of iterations becomes too large, 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 only 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.1) is specialized with respect to the number of data points (np). It contains ten loops with several degrees of nesting. One of these loops, with complexity $O(np^3)$, 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 6 (e)). However, in the case of program specialization, loop unrolling degrades performance. In contrast, data specialization produces a more stable speedup than the speedup due to program specialization regardless of the number of data points. When np is smaller than 512, data specialization does not obtain a better result in comparison to program specialization. However, when np 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 and small loops, combined specialization, where the loop which manipulates the more expensive computations to be cached is not unrolled, improves performance more than data specialization alone.

The Romberg integration (Appendix A.2) is specialized with respect to the number of iterations (M) used in the approximation. The Romberg integration contains two calls to the costly function *int_pow*. It is called twice: once in a singly nested loop and another time in a doubly nested loop. The time complexity of the Romberg integration is $O(M^2)$. Because both specialization strategies eliminate these expensive library calls, the speedup is consequently good. As for the FFT, too much loop unrolling causes the speedup due to program specialization to decrease, whereas combined specialization improves performance better than data specialization alone (Figure 6 (f)).

4.3. Instruction and data cache

To explain the benefit of the combined specialization, we have analyzed the effect of the code explosion on the Romberg integration and measured the instruction and data cache misses.

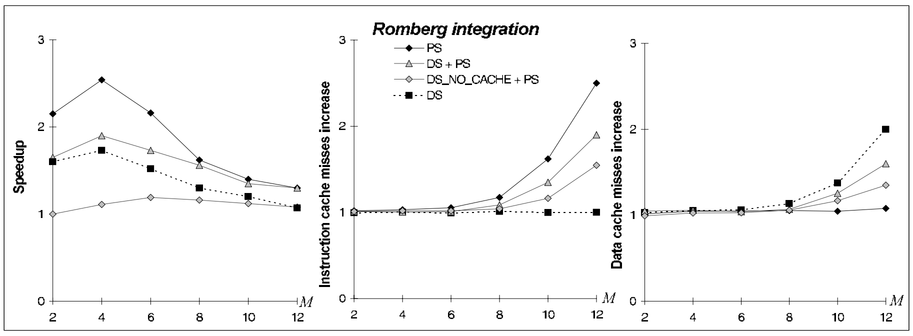

Figure 7. Instruction and data cache misses

Figure 7 shows that for the Romberg integration, combining program and data specialization reduces the instruction cache misses with regards to program specialization, and also reduces the data cache misses with regards to data specialization. To show that the profit of speedup is not just due to the loops not being unrolled, we have included the measurement of the combined specialization without frontier term caching. The figure shows how the instruction cache misses can dramatically reduce the performance of programs produced using program specialization. We conclude that a better balance between instruction and data cache can improve the performance of the specialized program.

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 sizes. However, as the problem size increases, the residual program produced by program specialization may become very large and often slower than the unspecialized program. On the other hand, data specialization can handle large problem sizes without much performance degradation. This strategy can, however, be ineffective if the program to be specialized consists mainly of control flow computations. The combination of both program and data specialization is promising: it can produce a residual program that is more efficient than with data specialization alone, without dropping in performance as dramatically as with program specialization, when the problem size increases.

Acknowledgments

We would like to thank 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 the Berkeley packet filter [17] and Julia Lawall on the fast Fourier transformation [12].

We wish to thank Olivier Danvy and the anonymous referees for their helpful comments on previous drafts of this paper.

Notes

1. Tempo, an off-line program specializer for C programs, is publicly available: http://www.irisa.fr/compose/tempo

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, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation*, 30(3), 1998.
4. C. Consel, L. Hornof, F. Noel, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.
5. R. Glück, R. Nakashige, and R. Zöchling. Binding-time analysis applied to mathematical algorithms. In J. Doležal and J. Fidler, editors, *System Modelling and Optimization*, pages 137–146. Chapman & Hall, 1995.

6. B. Guenter, T.B. Knoblock, and E. Ruf. Specializing shaders. In *Computer Graphics Proceedings*, Annual Conference Series, pages 343–350. ACM Press, 1995.
7. L. Hornof. *Static Analyses for the Effective Specialization of Realistic Applications*. PhD thesis, Université de Rennes I, June 1997.
8. L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. *Theoretical Computer Science*, 1999. to appear.
9. 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.
10. N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.
11. 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.
12. J.L. Lawall. Faster Fourier transforms via automatic program specialization. In *Partial Evaluation—Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, Copenhagen, Denmark, July 1998.
13. K. Malmkjaer. Program and data specialization: Principles, applications, and self-application. Master's thesis, DIKU University of Copenhagen, August 1989.
14. 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.
15. 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.
16. F. Noel, 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.
17. S. Thibault, L. Bercot, C. Consel, R. Marlet, G. Muller, and J. Lawall. Experiments in program compilation by interpreter specialization. Research Report 3588, INRIA, Rennes, France, December 1998.
18. S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, Indiana, October 1998.

Appendix

For each programs, the early inputs used for specialization in section 4 are shown in bold. The data computations cached by data specialization are shown in bold too.

A.1. Fast Fourier transformation

```
#define PI 3.14159265358979323846

int fft(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,xt;
  double  r1,r2,s1,s2,s3;
  double  cc1,ss1,cc3,ss3;

  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;i0++)
        {
          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;i0++)
    {
      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);
}
```

A.2. Romberg integration

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

A.3. 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;
}
```

A.4. Smirnov integration

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

  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++)
  {
    W=(double)i/(double)(i+n);
    suif_tmp0=u;
    u[0]=u[0]*W;
    if (c<(double)(n*i))
      u[0] = 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];
}
```

A.5. Cubic spline interpolation

```
void csi (int n,float x[100],
          float y[100],float z[100])
{
  float h[100],b[100]
  float u[100],v[100];
  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;
  while (i>=1)
  {
      z[i]=(v[i]-h[i]*z[i+1])
          /u[i];
      i=i-1;
  };
  z[0]=0;
}
```

A.6. Berkeley packet filter

```
/*-
 * Copyright (c) 1990, 1991, 1992,
 *           1993, 1994, 1995, 1996
 * The Regents of the University
 *                  of California.
 * All rights reserved.
 */
/*The instruction encondings.*/
#define BPF_LD 0x00
#define BPF_LDX 0x01
#define BPF_ST 0x02
#define BPF_STX 0x03
#define BPF_ALU 0x04
#define BPF_JMP 0x05
#define BPF_RET 0x06
#define BPF_MISC 0x07
#define BPF_W 0x00
#define BPF_H 0x08
#define BPF_B 0x10
#define BPF_IMM  0x00
#define BPF_ABS 0x20
#define BPF_IND 0x40
#define BPF_MEM 0x60
#define BPF_LEN 0x80
#define BPF_MSH 0xa0
#define BPF_ADD 0x00
#define BPF_SUB 0x10
#define BPF_MUL 0x20
#define BPF_DIV 0x30
#define BPF_OR 0x40
#define BPF_AND 0x50
#define BPF_LSH 0x60
#define BPF_RSH 0x70
#define BPF_NEG 0x80
#define BPF_JA 0x00
#define BPF_JEQ 0x10
#define BPF_JGT 0x20
#define BPF_JGE 0x30
#define BPF_JSET 0x40
#define BPF_K 0x00
#define BPF_X 0x08
#define BPF_A 0x10
#define BPF_TAX 0x00
#define BPF_TXA 0x80
/* Number of scratch memory words
 * (for BPF_LD|BPF_MEM and BPF_ST).*/
#define BPF_MEMWORDS 16
struct bpf_insn { u_short code;
                  u_char  jt;
                  u_char  jf;
                  int     k; };

u_int bpf(pc,p,wirelen,buflen)
  struct bpf_insn *pc;
  u_char *p;
  u_int wirelen,buflen;
{
  u_int A,X;
  int k,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) {
    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;
      default:
         returned=TRUE;
         abort();
         break;
      }
   }
   return 0;
}
```