Partial Evaluation of Numerical Programs in Fortran

Romana Baier, Robert Glück[1], Robert Zöchling
University of Technology Vienna, Institut für Computersprachen,
Argentinierstraße 8, A-1040 Vienna, Austria
E-mail: e1802gab@vm.univie.ac.at

Abstract

We investigate the application of partial evaluation to numerically oriented computation in scientific and engineering applications. We present our results using the Fast Fourier Transformation, the N-body attraction problem, and the cubic splines interpolation as examples. All programs are written in Fortran 77, specialized using our Fortran partial evaluator, and compiled into executable machine code using a commercial Fortran compiler. The results demonstrate that existing partial evaluation technology is strong enough to improve the efficiency of a large class of numerical programs. However, using partial evaluation as a development tool in the 'real world' still remains a challenging problem.

1 Introduction

Numerically oriented computation is a large application area where high performance is extremely important. Many applications in engineering, physics and other natural sciences critically depend on efficient numerical computations and even a small speedup can make a big difference in practice. But programmers face a dilemma: on the one hand they want to write general and well-structured programs that are easy to maintain, on the other hand specialized programs solving particular problems are often significantly faster, but are time-consuming to write and much harder to understand.

Partial evaluation is a method to automatically overcome losses in performance which are due to general algorithms [10,15]. Partial evaluation can improve the efficiency of programs by exploiting known information about the input of a program. For example, in the Fast Fourier Transformation the sampling rate can be fixed while the signal varies.

Despite the successful application of partial evaluation to declarative languages, such as Scheme or Prolog, only few attempts have been made to study partial evaluation of imperative languages. Our goal was to extend previous work on partial evaluation of imperative languages and to obtain more results in an important application area. In this paper we apply partial evaluation to numerically oriented problems. We present our results using the Fast Fourier Transformation, the N-body attraction problem, and the cubic splines interpolation as examples.

Our work differs from most previous works in that we apply partial evaluation in a more realistic setting. We use Fortran that, after all, is the most widely used language for scientific and engineering applications [11], a commercial Fortran compiler, and the source code for the Fast Fourier Transformation was taken from a Fortran compiler benchmark. Furthermore, we implemented a partial evaluator for a subset of Fortran that is based on current partial evaluation technology [16]. This extends previous works on partial evaluation of imperative languages and of numerical computations.

Why Fortran? We chose this imperative language for several reasons. There is still a large gap between the state-of-the-art of current research in partial evaluation and its potential significance. Due to the widespread use of Fortran over a period of more than 30 years, a vast body of expertise is available in the form of standard libraries representing an enormous amount of human effort and investment. It is unlikely that existing applications and libraries will be ported to other languages in order to take full advantage of partial evaluation.

The rest of this paper is organized as follows. Section 2 discusses partial evaluation and numerical programs. Section 3 provides a short overview of the Fortran partial evaluator system. Examples and results are given in Section 4. The performance of the partial evaluator is shown in Section 5. Related work is discussed in Section 6. Finally, Section 7 gives conclusions and directions for future work.

The Appendices A-B contain the source and the residual program of the Fast Fourier Transformation and the cubic splines interpolation. We assume a basic knowledge about partial evaluation (for a comprehensive discussion see e.g. [15]).

2 Partial Evaluation and Numerical Computation

Partial evaluation is most effective in cases when parts of the input information change less frequently than others, and it gives substantial savings when a residual program is used many times. This is the case in many numerically oriented problems. For example, in applications such as *circuit simulation* where multiple sets of input data and initial conditions need to be run repeatedly before the modelled system changes, or in *signal-processing* systems where numerical simulations run for a long time. However, on smaller problems, the time spent in partial evaluation may exceed the time saved by the optimizations, or the code size is increased dramatically. So, one has to consider whether a numerical problem (or which part of the problem) is appropriate for using partial evaluation.

[1] Supported by the Austrian Science Foundation (FWF) under grant number J0780 & J0964. Current address: DIKU, Dept. of Computer Science, Univ. of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark. E-mail: glueck@diku.dk

Data-Dependent vs. Data-Independent Computation

A characteristic feature of numerically oriented programs is that most of their numerical computations require dynamic data (signals, measurements) and therefore cannot be computed during specialization. But in many cases the control flow can be determined at specialization time. For example, for any given matrix size, matrix-multiply performs a fixed set of multiplications, regardless of the numerical values of the elements being multiplied.

Numerical programs can be divided into data-independent and data-dependent code sequences. A sequence of operations is *data-independent* if the control flow can be determined at specialization (compile) time. Data-independence is a necessary attribute to predict what operations a program will perform next. Being able to predict this makes partial evaluation more effective: iterative loops can be unfolded and conditionals can be reduced to one of the branches.

Partial evaluation works best for data-independent control flow in programs. It speeds up the run time of numerical computations that operate on static data and eliminates static arguments of procedures and functions. In case all arguments of a procedure or function call are static, the call can be executed at specialization time and hence it is eliminated from the residual program.

Partial evaluation cannot determine the control flow when applied to *data-dependent* computations. For example, if the test of a conditional `IF` depends on dynamic values then the test can not be decided and code has to be generated for both branches. Partial evaluation is not effective when computations are extremely data-dependent. For example, in techniques for *linear programming* the choice of pivot is not known before run time, but depending on this choice different computations have to be performed.

Fortunately, in many applications of numerical programs, data-independent regions are extremely large. Data-dependent conditionals occur preferably at the end of long computations for operations such as convergence checks and strategy selection. This opens a wide applicability for partial evaluation in numerically oriented computations.

Limitations

As mentioned above partial evaluation unfolds static loops and generates code for the then- and the else-branch of dynamic conditionals. This may result in large residual programs. For that reason it is not always desirable to unfold static loops at specialization time. When the number of iterations is large the loop ought to be left intact when the corresponding body contains many dynamic statements. This has to be considered in the preprocessing phase during the binding-time analysis. It can be a serious problem especially in numerically oriented programs where the number of iterations is rather large.

For example, in particle physics the attraction of particles in a system is of interest. The number of particles to be considered can become extremely large. The algorithm for solving this problem, the N-body problem (see Section 4.2), contains several static iterations depending on the number of particles.

Partial evaluation of numerical programs could further benefit from traditional compiler optimizations such as the use of algebraic identities for the reduction of expressions.

3 Partial Evaluator for Fortran 77

We implemented a partial evaluator for a substantial subset of Fortran 77. In this section we give a short overview of the system. More details and results for non-numerical applications can be found in [16].

The Subset of Fortran 77

We selected a subset of Fortran 77 which is large enough to include characteristic features of the language [1,11], and at the same time small enough to allow the development and full implementation of the partial evaluator.

The subset of the language, called F77, includes multi-dimensional arrays, functions, procedures, and `COMMON` regions (to specify global storage accessible within procedures and functions). The statements include arithmetic assignments, nested conditionals `IF`, unconditional jumps `GOTO`, procedure calls `CALL`, function calls, the `DO`, the `RETURN`, and the `CONTINUE` statement. Expressions include constants, identifiers, indexed arrays, arithmetic and relational operators. Simple input/output facilities are supported. As in Fortran 77 all parameters are passed *call-by-reference* to functions and procedures (i.e. they may be modified in a function or procedure).

We require that F77 programs are ‘well-typed’, that is, the dimensions of formal and actual array parameters in procedure and function calls must match (Fortran 77 allows to pass, say, a 2-dimensional array as argument to a parameter declared as a 3-dimensional array). The subset does not include the Fortran `EQUIVALENCE` statement for sharing storage units (this statement instructs the compiler to arrange for variables and arrays to access the same physical storage). The syntax of F77 is defined in [16].

Partial Evaluator System

The input and output of the partial evaluator are programs written in F77. The partial evaluator is *off-line*, meaning that before a source program is specialized with respect to the given input, it is binding-time analyzed. Source and residual programs can be compiled directly into executable machine code using any Fortran 77 compiler. The partial evaluator itself is fully implemented in Fortran 77 and is therefore highly portable. The system has three main phases (Figure 1).

- The *preprocessing phase* translates an F77 source program into an intermediate language, called CoreF. The binding-time analysis (BTA) annotates all statements (expressions) in the source program as either static or dynamic corresponding to the static/dynamic classification (S/D) of its input. The output of the preprocessing phase is an annotated CoreF program.

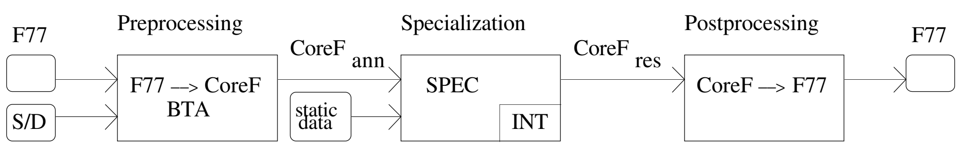

Figure 1: The structure of the Fortran partial evaluator.

- The *specialization phase* takes the annotated CoreF program and the static data as input and specializes the program with respect to the static data. This phase, the kernel of the partial evaluator system, contains an interpreter (INT) for the evaluation of static CoreF statements. The output of the specialization phase is a specialized CoreF program.

- The *postprocessing phase* translates a specialized CoreF program into F77.

A short overview of the binding-time analysis and the specialization phase follows. The translators between F77 and CoreF use conventional methods and are not described here.

Binding-Time Analysis (BTA)

The binding-time analysis (BTA) annotates each statement (expression) in a source program as either static or dynamic. The task of the BTA is to determine a congruent division of the variables in the source program, that is, variables classified as static may depend only on static values. The analysis used in the BTA is *monovariant* (every statement (expression) can only be given one static/dynamic classification) [15]. The BTA is implemented by a fixed-point iteration: an approximative algorithm which iterates over the static-dynamic division until a stable classification is reached.

The output of the BTA is an annotated CoreF program. To represent static/dynamic annotations a *two-level* version of CoreF is used in which every keyword appears in either of two versions: in a normal version (=static) and in an underlined version (=dynamic).

Partial Evaluation of Statements

The specialization phase follows the annotations made by the binding-time analysis: it executes static statements, reduces partially static expressions, and specializes dynamic program points (functions, procedures, basic blocks).

During the specialization of a dynamic basic block the partial evaluator runs through the sequence of statements step-by-step, executing static statements and generating code for dynamic ones (shown in pseudo-code in Figure 2). When a dynamic conditional, e.g. an `IF`, is met, both branches are specialized. The static version of the `RETURN` and `END` statement does not appear in Figure 2 because they occur only in fully static functions and procedures, and therefore they are handled by the CoreF interpreter in the partial evaluator.

A *polyvariant program point specialization* is used for the specialization of dynamic program points [9,15]: the same program point may be specialized with respect to different static storages. Program point specialization includes function and procedure specialization, and specialization of target points for jumps. The specialization of dynamic functions and procedures is *depth-first*. It is necessary to specialize a function or procedure before the statements following the call are specialized, because of the effects a procedure or function can have on the static storage. A *done-* and a *pending-list* are used to keep track of already specialized program points and program points pending to be specialized.

4 Experiments

We used the Fortran partial evaluator for a number of numerically oriented problems in scientific and engineering applications. We present our results using the Fast Fourier Transformation, the N-body problem, and the cubic splines interpolation. Each problem is described and the results are discussed.

```
repeat
 switch(stmt_at(label))
  case ASSIGN:    <evaluate expression & update lval>;
                  label=next(label);
  case ASSIGN:    gen_assign(<reduce expression>);
                  label=next(label);
  case IF:        if (<evaluate expression>)
                  then label=<then label>
                  else label=<else label>;
  case IF:        gen_if(<reduce expression>);
                  add_pending(<then label>);
                  add_pending(<else label>);
                  stop=TRUE;
  case SCALL:     <perform static procedure>;
                  label=next(label);
  case SCALL:     gen_scall(<reduce expressions>);
                  <specialize procedure>;
                  label=next(label);
  case FCALL:     <perform static function
                  & update lval>;
                  label=next(label);
  case FCALL:     gen_fcall(<reduce expressions>);
                  <specialize function>;
                  label=next(label);
  case GOTO:      label=<goto label>;
  case GOTO:      gen_goto();
                  add_pending(<goto label>);
                  stop=TRUE;
  case CONTINUE: label=next(label);
  case CONTINUE: gen_continue();
                  label=next(label);
  case RETURN:    gen_return(); stop=TRUE;
  case END:       gen_end(); stop=TRUE;
until (stop);
```

Figure 2: Code generation for statements in the specialization phase.

All source and residual programs are written in a subset of Fortran 77 (Section 3). The *run times* are given in user seconds using the Lahey Fortran compiler (version 5.01) [17] and an IBM AT386/25MHz. The *size* of the program is given as *lines* of 'pretty-printed' Fortran source code and *KBytes* of machine code.

The Appendices A-B contain the source and the residual program of the Fast Fourier Transformation and the cubic splines interpolation. The programs for the N-body problem are too large to be shown here.

4.1 Fast Fourier Transformation

The *Fourier Transformation* is used in many numerical applications, e.g. in electrical engineering and physics. The discrete Fourier Transformation approximates the Fourier Transformation by sampling an input signal *n*-times. The *Fast Fourier Transformation* (FFT) is the fastest known algorithm for calculating a discrete Fourier Transformation [8].

The FFT is a data-independent numerical program in which a significant number of numerical operations is independent of the dynamic data (input signals, measurements).

Program and Results

The inputs of the FFT source program are a continual function (given as a sequence of discrete input signals) and a variable *n* indicating the sampling rate of the function. The program returns the transformed input function approximated by *n* values. In most cases a standard sampling rate is used and only the input function varies. For a huge sampling rate the execution time is important. One may speed up the performance of the FFT by specializing the source program with respect to a fixed sampling rate *n*.

FFT problem	Run time	Speedup	Lines	KBytes
Source (n=10)	0.64		179	40.7
Residual	0.16	4.0	497	44.2
Source (n=50)	4.23		179	40.7
Residual	1.31	3.2	3881	89.3
Source (n=100)	9.89		179	40.7
Residual	3.19	3.1	9309	162.3

The specialization of the source program to a static sampling rate *n* yields a residual program with a completely data-independent control flow. The relatively high speedup between 3.1 and 4.0 is achieved because all iterations can be unfolded and all conditionals can be eliminated. In addition, most of the numerical operations depend only on the static sampling rate *n* and can be computed at specialization time. For larger sampling rates the speedup slowly decreases due to a large residual program. The higher the sampling rate is, the more often specialized versions of the functions `Rcmul` (multiply real part) and `Icmul` (multiply imaginary part) are generated during specialization (see Appendix A).

The source program of the FFT was taken from a Fortran compiler benchmark and adapted for the Fortran subset of our partial evaluator. We made the following changes in the benchmark code in order to adapt it to the source language of our partial evaluator: we simulated complex numbers by an array containing the real and imaginary part of a complex number and we omitted the calculation of the input function (and used pre-computed values of the function instead). As input we used the ramp function. For all measurements the program was executed 100 times.

For any given *n* three out of six procedures in the source program can be eliminated due to static arguments. For n=10 this elimination saves 75 out of 104 calculations of procedures in the residual program. All 163 conditionals are eliminated and 446 numerical operations out of 572 can be performed at specialization time. Further 40 loops are unfolded which saves all 214 evaluations of loop conditions.

4.2 N-body Problem

The *N-body attraction problem* involves computing the trajectories of a collection of *N* particles which exert forces on each other. In order to simulate a future motion of a particle, the program integrates the forces over time. The N-body problem arises in many scientific applications, such as particle physics, astronomy, and space travel.

In astronomy the 6-body and 9-body problem are of particular interest. Our solar system is a 10 particle system in which the forces are due to gravitational attraction. The 6-body problem includes only the outer planets and the sun, allowing to investigate the long-term stability of the solar system. The 9-body problem describes the motion of the solar system, excluding Mercury (because its high eccentricity necessitates the use of an extremely small integration step-size that makes long-term integrations impractical).

Almost all numeric computations in the N-body problem depend on dynamic numeric data (positions and velocities), but the control flow of the program is entirely data-independent. This is the case in many numerical applications.

Program and Results

The inputs of the N-body source program are an initial state of our solar system, a variable *N*, a time-step, and a number of future states to be computed. The initial state contains masses, positions, and velocities of the planets at a certain point in time. The variable *N* defines the number of planets to be considered. The time-step specifies the difference in time between two consecutive system states. The program returns the desired future states. In particle physics the variable *N* may be extremely large which yields to a long execution time. In this case each gain in performance is of great interest.

We assumed the number *N*, the time-step, and the masses of the planets to be known before run time. The dynamic input of the program are the positions and velocities of the planets at a certain point in time. For any given *N*, the N-body problem is entirely data-independent. This means that in the residual program all conditionals are eliminated and all loops unfolded. The result is a residual program without any conditionals that may be very large. Compared to other experiments the savings in this example are relatively low resulting in a speedup of 1.4. Most of the expansive numerical operations depend on dynamic data and have to be performed at run time.

The source program of the N-body problem was adapted from a program originally written in Scheme [4]. The program uses the *Runge-Kutta integrator* to perform the integration-step. Some changes had to be made due to the different language paradigms of Fortran and Scheme. We used arrays as data structures instead of lists in Scheme. Procedures performing numerical operations on all elements of a data structure are implemented as loops. Further in the original program a function is passed as an argument. High-order features are not available in Fortran 77 and we imitate this by passing an `INTEGER` number identifying a certain function. Altogether the Fortran program is more efficient, and contains fewer procedures and functions as the original Scheme program.

N-body problem	Run time	Speedup	Lines	KBytes
Source (N=6)	3.41		378	43.9
Residual	2.42	1.4	3361	84.0
Source (N=9)	6.92		378	43.9
Residual	5.21	1.3	7233	133.4

The input used in the experiments is the same as suggested in [6]. For all measurements we computed 200 future states. To produce one future state of the solar system (assuming that N=6) all 64 conditionals can be eliminated in the residual program and due to the known masses 216 numerical operations out of 2338 can be performed before run time. Further 22 loops are unfolded which saves all 552 evaluations of loop conditions.

Berlin and Weise [6] report a speedup of 38 for the same program written in Scheme using a prototype compiler into C that provides also partial evaluation. This is far better. However, a direct comparison is difficult because of the different languages that are involved, and the combination of compilation with partial evaluation.

Their prototype compiler uses a placeholder-based symbolic execution technique to perform partial evaluation of Scheme [4]. During partial evaluation the type of an input data structure, represented as a placeholder, is assumed to be static which allows an efficient implementation of the data structure in the final residual program. In Fortran, however, the data type is always 'static' due to the required type declarations. Therefore efficient data structures can be generated by the Fortran compiler for both, the source and the residual program. The same Fortran compiler was used to compile the source and the residual program directly into executable machine code and it uses other optimizations than the Scheme compiler.

Another difference is due to the fact that their system propagates the mass of Pluto, approximated by 0, throughout the source program eliminating numerous computations thereby. Although this operation is conceptually simple, it is difficult in an off-line partial evaluator [15], because the BTA cannot predict that the result of the multiplication will be static. However, much of the effect can be obtained by a postprocessing phase or by an optimizing compiler.

4.3 Cubic Splines Interpolation

Interpolation is an important numerical method used in a wide range of scientific and engineering applications. The *cubic splines interpolation* approximates the values between two measured values with a cubic polynomial. To avoid 'edges' two consecutive polynomials must have the same values and derivatives at the point they meet. The intended result is a continuous function that connects all measured values.

Different range conditions restrict the possible behavior of the curve at the beginning and the end. (i) *Natural range conditions* make the bending in the endpoints zero, that means the second derivative disappears. (ii) *Periodical range conditions* imply the first and the last value to be equal and are used in case the function is known to be periodical. (iii) *De Boor range conditions* require the first and last two polynomials to be equal. In each of these cases the construction of the derivatives requires solving a *tridiagonal equation*. The cubic splines interpolation is used in algorithms for signal-processing, in real-time systems, and in graphical applications.

The cubic spline interpolation has a control-flow that is not entirely data-independent, but the data-dependent computations occur only at the end of the program. This is a typical situation in numerical algorithms where operations such as converenge checks occur preferably at the end of long data-independent computations.

Program and Results

The inputs of the cubic splines interpolation are the number of measured values, the x- and y-coordinates of these values, the distance between each two consecutive measured values, the x-coordinate of the value to be computed, and the type of range conditions. We implemented the cubic splines interpolation with all three types of range conditions. The program assumes an equal distance between two consecutive values. It returns the value of the y-coordinate at the desired x-coordinate.

In our experiments we made the assumption that the number of values, their distance and the type of range conditions (e.g. *type=periodical*) are known before run time. The x- and y-coordinates of the measured values and the x-coordinate of the value to be computed are assumed to be dynamic data.

By specializing the program to *natural* and *de Boor* range conditions a speedup of 4.0 is obtained. The speedup of the program specialized to *periodical* range conditions is 6.0. Knowing the number of values it is possible to unfold each iteration except the one located at the end of the program. Considering the information about the type of range conditions the program is specialized to the correct branch and three conditionals are saved. The result is a data-independent program up to the last iteration. For all measurements the program was executed 1000 times.

Cubic Splines problem	Run time	Speedup	Lines	KBytes
Source (natural)	1.76		226	34.6
Residual	0.44	4.0	164	32.6
Source (periodical)	2.64		226	34.6
Residual	0.44	6.0	172	32.7
Source (de Boor)	1.76		226	34.6
Residual	0.44	4.0	166	32.6

The most significant performance speedup results from the known distance between two values. This is why three more arguments of the tridiagonal equation solver (procedure *Tridia*) become static. The code in the procedure is entirely data-independent and most of the numerical computations in this procedure can be saved during run time. Instead of 85 only 26 statements are executed at run time. The source code of the procedure *Tridia* is shown in Figure 3 together with a specialized version.

For *natural* range conditions and 7 measured values 10 conditionals out of 14 can be eliminated in the residual program. Further 141 numerical operations (only 102 in Tridia) out of 255 can be performed before run time and 3 loops are unfolded which saves all 18 evaluations of loop conditions. This is also similar to *de Boor* range conditions.

Especially many computations can be saved if *periodical* range conditions are used. In this case the procedure *Tridia* is invoked twice. The second invocation of the procedure *Tridia* becomes fully static and the equation solver is executed by the interpreter at specialization time. Due to monovariant binding-time analysis a copy of the procedure (*Tridia1*) has to be used for partial evaluation. That is, the second invocation of the equation solver is not needed in the residual program (see Appendix B). Exactly 13 conditionals out of 17 can be eliminated in the residual program and 236 numerical operations (only 201 in Tridia and Tridia1) out of 357 can be performed at specialization time. Further 7 loops are unfolded which saves all 39 evaluations of loop conditions. The savings in this special case explain the higher speedup with *periodical* range conditions.

This example demonstrates that a general purpose procedure, in our example the procedure *Tridia*, can be specialized with respect to specific arguments resulting in a large speedup. If *Tridia* is executed with unknown coefficients the speedup would not be significant, but using the same code for calculating cubic splines, coefficients become static and numerical operations can be saved during run time. In case the range conditions are unknown at specialization time, the performance of the residual program will hardly be affected, but the residual program may become about three times larger because the code for all three types will have to be specialized.

The cubic splines interpolation illustrates the use of a partial evaluator to generate specialized versions of a generic algorithm. The tridiagonal equation solver in Figure 3 shows that it is easier to write and maintain a generic algorithm, and to automatically generate specialized versions with a partial evaluator than to implement and verify each specialized version by hand.

5 Performance of the Partial Evaluator

To illustrate the performance of the system, the run times of the binding-time analysis and the specialization phase are shown. The two translators between Fortran 77 and the internal language CoreF use conventional methods and are not discussed here. The

```
c Annotated source code of procedure Tridia.

    ....
    b(n)=0
    DO 10 i=1,n-1
     IF (c(i) .NE. 0) THEN
      t=a(i)/c(i)
      si=1/SQRT(1+t*t)
      co=t*si
      a(i)=a(i)*co+c(i)*si
      h=b(i)
      b(i)=h*co+a(i+1)*si
      a(i+1)=(-h)*si+a(i+1)*co
      c(i)=b(i+1)*si
      b(i+1)=b(i+1)*co
      hlp=sy(i)
      sy(i)=hlp*co+sy(i+1)*si
      sy(i+1)=(-hlp)*si+sy(i+1)*co
     END IF
10  CONTINUE
    sy(n)=sy(n)/a(n)
    hlp=b(n-1)*sy(n)
    sy(n-1)=(sy(n-1)-hlp)/a(n-1)
    DO 20 i=n-2,1,-1
     hlp=b(i)*sy(i+1)-c(i)*sy(i+2)
     sy(i)=(sy(i)-hlp)/a(i)
20  CONTINUE
    END
```

```
c Residual code of procedure Tridia.
c Tridia was specialized with respect to periodical range
c conditions, a fixed distance (h=1), and a fixed number of
c measured values (n=7).

    ....
    hlp = sy(1)
    sy(1) = (hlp*0.948683)+(sy(2)*0.316228)
    sy(2) = ((-hlp)*0.316228)+(sy(2)*0.948683)
    hlp = sy(2)
    sy(2) = (hlp*0.961074)+(sy(3)*0.276289)
    sy(3) = ((-hlp)*0.276289)+(sy(3)*0.961074)
    hlp = sy(3)
    sy(3) = (hlp*0.963174)+(sy(4)*0.268879)
    sy(4) = ((-hlp)*0.268879)+(sy(4)*0.963174)
    hlp = sy(4)
    sy(4) = (hlp*0.963408)+(sy(5)*0.268039)
    sy(5) = ((-hlp)*0.268039)+(sy(5)*0.963408)
    hlp = sy(5)
    sy(5) = (hlp*0.963431)+(sy(6)*0.267957)
    sy(6) = ((-hlp)*0.267957)+(sy(6)*0.963431)
    sy(6) = sy(6)/2.63214
    hlp = 1.73205*sy(6)
    sy(5) = (sy(5)-hlp)/3.73194
    hlp = (2.00009*sy(5))-(0.268039*sy(6))
    sy(4) = (sy(4)-hlp)/3.73080
    hlp = (2.00120 * sy(4))-(0.268879*sy(5))
    sy(3) = (sy(3)-hlp)/3.71915
    hlp = (2.01691*sy(3))-(0.276289*sy(4))
    sy(2) = (sy(2)-hlp)/3.61939
    hlp = (2.21359*sy(2))-(0.316228*sy(3))
    sy(1) = (sy(1)-hlp)/3.16228
    END
```

Figure 3: The tridiagonal equation solver *Tridia*.

times are tentative since the phases in the present version of the system were not optimized for speed (the numbers do not include the time required to read/write programs from disk).

Partial evaluation gives substantial savings when a residual program is used several times. But partial evaluation can even pay off in a single run if the specialization time plus the residual run time is faster than the run time of the source program.

Problem \ Phase	*BT Analysis*	*Specialization*
FFT (n=10)	0.77	3.35
FFT (n=50)	0.77	35.53
FFT (n=100)	0.77	123.64
6-Body	3.90	15.33
9-Body	3.90	50.37
Splines (natural)	1.75	0.82
Splines (periodical)	1.75	0.94
Splines (de Boor)	1.75	0.83

Note, that the annotated version of a source program can be reused as long as the static/dynamic classification of the input does not change. For example, in the case of the cubic splines interpolation the binding-time analysis needs to be performed only once (not three times). While the run time of the binding-time analysis depends on the size of the source program and the initial static/dynamic classification, the specialization phase depends on the annotated program and the static values. The run time of the specialization phase is determined by the size of the generated code and the number of static computations performed during specialization. Depending on the static values the generated residual program can be several times larger than the original source program (due to the polyvariant specialization). In our experience the binding-time analysis usually requires the smallest portion of the total run time.

6 Related Work

Early examples of specializing numerical algorithms are provided by Gustavson et al. [14] and Goad [13]. They do not associate themselves with the partial evaluation paradigm, however. More recently Berlin and Weise [6,4,5] applied partial evaluation to numerical programs written in Scheme.

Gustavson et al. describe a generator using symbolic processing for generating Fortran programs which represent the optimal reduced Crout algorithm. Their generator actually specializes the full Crout algorithm with respect to a fixed sparseness structure of an NxN matrix. The generation algorithm systematically exploits sparseness by eliminating unnecessary arithmetic operations and by storing information in a compact, directly accessible manner [14].

Goad proposes a method for the automatic construction of special purpose programs using symbolic execution. He uses a computational problem from three dimensional graphics, namely the hidden surface elimination problem, to derive different special purpose programs for fixed scenes and variable positions of the viewer [13].

Berlin and Weise applied partial evaluation to the N-body problem written in Scheme [6,4,5]. They used a prototype compiler incorporating partial evaluation which uses a placeholder-based symbolic execution technique to perform partial evaluation. Numerical values that are not available until run time are represented symbolically using a data structure known as a placeholder. The placeholders for input and output data are used to create appropriate data structures at compile time. The type of the input data structure is assumed to be static.

Despite the successful application of partial evaluation to declarative languages, such as Scheme or Prolog, only few attempts have been made to study partial evaluation in imperative languages. Ershov and his group were the first who investigated mixed computation and imperative languages [12,9]. Pagan describes manual methods for generating language processors in Pascal [19,20]. Meyer studied on-line partial evaluation for a Pascal-like language [18], allowing potentially more specializations during partial evaluation than an off-line specializer.

The application of partial evaluation to software maintenance is discussed by Blazy and Facon [7]. Their partial evaluator aims at improving the readability of Fortran programs, but not at improving the efficiency of the programs.

Andersen developed the first self-applicable partial evaluator for a substantial subset of C [2]. This partial evaluator handles recursive functions and procedures, multi-dimensional arrays, and pointers [3].

7 Conclusion and Future Work

We applied the partial evaluator to several numerically oriented programs with varying results. Some speedups may not seem impressively high compared to results with non-numerical applications achieved in declarative languages, as reported in the literature, but the improvements are definitely useful for numerical applications where high performance is extremely important. Our results demonstrate that existing partial evaluation technology is strong enough to improve the efficiency of a large class of numerically oriented problems in scientific and engineering applications.

In many cases the control flow can be determined at specialization time. Data-independent regions are extremely large in many numerical applications. Thus loops can be unfolded and conditionals can be eliminated. For example, the Fast Fourier Transformation can be specialized with respect to a fixed sampling rate, or the distance and number of values (signal, measurements) is known before run time as in the cubic splines interpolation.

Partial evaluation enables to automatically tailor generic numerical programs to specific needs and applications. This opens a possibility for implementing general algorithms without loss of efficiency. For example, in the case of the cubic splines interpolation the tridiagonal equation solver can be specialized with respect to a certain type of range conditions. Another promising direction is to exploit the parallelism exposed by partial evaluation on parallel computing systems, e.g. by using Fortran parallelizers.

But there is still room for improvements. Future work is desirable in several directions. Partial evaluators for numerical programs should take advantage of additional knowledge about mathematical and scientific functions, and exploit algebraic simplifications. For example, in the N-body problem a lot of computations could be saved by propagating a zero value throughout a series of numerical operations. Additional precision could be gained by using on-line partial evaluation techniques. We expect that these techniques will improve the results presented in this paper.

Using partial evaluation as a software development tool remains a challenging problem. 'Industrial-strength' partial evaluators will have to tackle a variety of existing programs and libraries. To obtain good results one will need to deal with various language features and programming styles. For example, passing arrays of different dimensions as arguments, or sharing the same physical storage is possible in Fortran, but, obviously, complicates the analysis and specialization of programs.

Another open question is the interaction of partial evaluation techniques with traditional compiler optimizations. The application of one transformation may invalidate conditions for applying another transformation. For example, loop unfolding by a partial evaluator may disable invariant code motion by an optimizing compiler. Although optimizing transformations have been used in compilers for a long time, their interaction with partial evaluation techniques and the order of applying them is not fully understood. More should be known about the impact of a particular computer hardware on the enabling/disabling conditions of optimizing transformations. Without a formal, concise way to specify the necessary conditions and actions, it will be difficult to ensure desirable optimization effects in practice.

Acknowledgments

We would like to thank Peter Kowatsch and Wolfgang Lair from *MSB Software* Vienna for their technical assistance and for many valuable discussions about the 'real world'. Special thanks are due to Hans Moritsch for his help with Fortran benchmarks, to Lars Ole Andersen and Kristian Nielsen for comments on an earlier version of the paper.

References

1. American National Standards Institute, *Programming language FORTRAN: approved April 3, 1978.* ANSI/X3.9-1978, New York 1978.
2. Andersen L. O., Partial evaluation of C and automatic compiler generation. In: Kastens U., Pfahler P. (ed.), *Compiler Construction. 4th International Conference*. (Paderborn, Germany). Lecture Notes in Computer Science, Vol. 641, 251-257, Springer-Verlag 1992.
3. Andersen L. O., Binding-time analysis and the taming of C pointers. In: *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. (Copenhagen, Denmark). 47-58, ACM Press 1993.
4. Berlin A., A compilation strategy for numerical programs based on partial evaluation. MIT AI Laboratory, Cambridge, Massachusetts. Technical Report No. TR-1144, 1989.
5. Berlin A., Partial evaluation applied to numerical computation. In: *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*. (Nice, France). 139-150, ACM Press 1990.
6. Berlin A., Weise D., Compiling scientific code using partial evaluation. In: *IEEE Computer*, 23(12): 25-37, 1990.
7. Blazy S., Facon P., Partial evaluation for the understanding of Fortran programs. In: *Software Engineering and Knowledge Engineering*. (San Francisco, California). 517-525, 1993.
8. Brigham E. O., *FFT Schnelle Fourier-Transformation*. R. Oldenburg: München, 2.Auflage, Wien 1985.
9. Bulyonkov M. A., Polyvariant mixed computation for analyzer programs. In: *Acta Informatica*, 21: 473-484, 1984.

10. Consel C., Danvy O., Tutorial notes on partial evaluation. In: *Conference Record of the Twentieth Symposium on Principles of Programming Languages*. (Charleston, South Carolina). 493-501, ACM Press 1993.
11. Ellis T. M. R., *Fortran 77 Programming: with an Introduction to the Fortran 90 Standard (2nd Edition).* International Computer Science Series. Addison-Wesley: Wokingham, England 1990.
12. Ershov A. P., Mixed computation: potential applications and problems for study. In: *Theoretical Computer Science*, 18: 41-67, 1982.
13. Goad C., Automatic construction of special purpose programs. In: Loveland D. W. (ed.), *6th Conference on Automated Deduction*. (New York, USA). Lecture Notes in Computer Science, Vol. 138, 194-208, Springer-Verlag 1982.
14. Gustavson F. G., Liniger W., Willoughby A. R., Symbolic generation of an optimal Crout algorithm for sparse systems of linear equations. In: *Journal of the ACM*, 17(1): 87-109, 1970.
15. Jones N. D., Gomard C. K., Sestoft P., *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series in Computer Science. Prentice Hall: New York, London, Toronto 1993.
16. Kleinrubatscher P., Kriegshaber A., Zöchling R., Glück R., Fortran program specialization. In: *Workshop on Semantikgestützte Analyse, Entwicklung und Generierung von Programmen*. (to appear), Justus-Liebig-Universität Giessen, Germany 1994.
17. Lahey Computer Systems, Inc., Lahey Fortran 77 Manual. Revision A. P.O. Box 6091, Incline Village, NV 89450-6091, 1991.
18. Meyer U., Techniques for partial evaluation of imperative languages. In: *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. (New Haven, Connecticut). 94-105, ACM Press 1991.
19. Pagan F. G., Converting interpreters into compilers. In: *Software - Practice and Experience*, 18: 509-527, 1988.
20. Pagan F. G., *Partial Computation and the Construction of Language Processors.* Prentice Hall Software Series. Prentice Hall: Englewood Cliffs, NJ 1991.

Appendix A
The Fast Fourier Transformation

A.1 Source Program

```
c Content    : Fast Fourier Transformation
c Source     : FFT Benchmark adapted by
c              Romana Baier, Robert Zöchling

c local variables
      REAL y, x
      DIMENSION y(100,2), x(100,2)
c y() indicates the ramp function, dynamic
      INTEGER n, incrm, nx, i, j, Log2
      INTEGER Perm
      INTEGER repeat, start, end

      CALL TIMER(start)
c n indicates the number of points, static
      n=10
      repeat=1
1     IF (repeat .NE. 100) THEN
      . . .
c initialize the real part and the imaginary part
c of y( ) by using the ramp function:
c     slope=1.0/(n-1)
c     do i=0,n-1
c       y(i)=CMPLX(i*slope,0.0)
c     end do
      . . .
c start FFT algorithm
      incrm=n/2
      nx=2
      i=Log2(n)-1
      DO 100 j=i,0,-1
          CALL Fftc2(y,nx,incrm)
          nx=nx*2
          incrm=incrm/2
100   CONTINUE
      DO 200 i=0,n-1
          j=Perm(i,n)+1
          x(i+1,1)=y(j,1)
          x(i+1,2)=y(j,2)
200   CONTINUE
      DO 300 i=1,n
          y(i,1)=x(i,1)
          y(i,2)=x(i,2)
300   CONTINUE
c output of the result: real + imaginary part of y( )
      ....
      repeat=repeat+1
      GOTO 1
      END IF
      CALL TIMER(end)
      print*,'Run time (n=10):',end-start
      END

      SUBROUTINE Fftc2(y,nx,incrm)
      REAL y
      DIMENSION y(100,2)
      INTEGER nx, incrm, evenp, oddp
      INTEGER  minptr, maxptr, pairl
      INTEGER indx, Perm, i
      REAL oddv, evenv, term2, cexpon
      DIMENSION oddv(2), evenv(2)
      DIMENSION term2(2), cexpon(2)
      REAL pi, fac, Rcmul, Icmul
c set pi to 3.14
      pi=3.14
      minptr=0
      maxptr=minptr+nx/2
      DO 400 pairl=minptr,maxptr-1
          indx=Perm(pairl,nx/2)
          fac=2.0*pi*indx/nx
          cexpon(1)=COS(fac)
          cexpon(2)=SIN(fac)
          evenp=pairl*2*incrm
          oddp=evenp+incrm
          DO 500 i=0,incrm-1
              evenp=evenp+1
              oddp=oddp+1
              oddv(1)=y(oddp,1)
              oddv(2)=y(oddp,2)
              evenv(1)=y(evenp,1)
              evenv(2)=y(evenp,2)
              term2(1)=Rcmul(cexpon,oddv)
              term2(2)=Icmul(cexpon,oddv)
              y(oddp,1)=evenv(1)-term2(1)
              y(oddp,2)=evenv(2)-term2(2)
              y(evenp,1)=evenv(1)+term2(1)
              y(evenp,2)=evenv(2)+term2(2)
500       CONTINUE
400   CONTINUE
      END

      REAL FUNCTION Rcmul(num1,num2)
c returns real part of a complex multiplication
      REAL num1, num2
      DIMENSION num1(2), num2(2)
      REAL hlp1
      hlp1=num1(1)*num2(1)
      Rcmul=num1(2)*num2(2)
      Rcmul=hlp1-Rcmul
      RETURN
      END
```

```
      REAL FUNCTION Icmul(num1,num2)
c returns imaginary part of a complex multiplication
      REAL num1, num2
      DIMENSION num1(2), num2(2)
      REAL hlp1
      hlp1=num1(1)*num2(2)
      Icmul=num1(2)*num2(1)
      Icmul=hlp1+Icmul
      RETURN
      END

      INTEGER FUNCTION Perm(l,m)
      INTEGER l, m
      INTEGER numbit, k, power2, Log2
      INTEGER Isodd, i
      numbit=Log2(m)
      k=l
      power2=m/2
      Perm=0
      DO 600 i=1,numbit
          IF (Isodd(k) .EQ. 1) THEN
              Perm=Perm+power2
          END IF
          power2=power2/2
          k=k/2
600   CONTINUE
      RETURN
      END

      INTEGER FUNCTION Isodd(n)
      INTEGER n
      IF (2*(n/2) .EQ. n) THEN
          Isodd=0
      ELSE
          Isodd=1
      END IF
      RETURN
      END

      INTEGER FUNCTION Log2(n)
      INTEGER n, k, i
      k=n
      Log2=0
      DO 700 i=1,n
          IF (k .GT. 1) THEN
              Log2=Log2+1
              k=k/2
          ELSE
              RETURN
          END IF
700   CONTINUE
      END
```

A.2 Residual Program

```
c specialized version of procedure Main
REAL y, x
DIMENSION y(100,2), x(100,2)
INTEGER repeat, start, end

      CALL TIMER(start)
      repeat=1
1     IF (repeat .NE. 100) THEN
          . . .
c initialize the real part and the imaginary part
c of y( ) as in the source proram
          . . .
c call procedure Fftc2
          CALL S2 (y)
c call procedure Fftc2
          CALL S3 (y)
c call procedure Fftc2
          CALL S4 (y)
          x(1,1) = y(1,1)
          x(1,2) = y(1,2)
          x(2,1) = y(6,1)
          x(2,2) = y(6,2)
          x(3,1) = y(3,1)
          x(3,2) = y(3,2)
          x(4,1) = y(8,1)
          x(4,2) = y(8,2)
          x(5,1) = y(2,1)
          x(5,2) = y(2,2)
          x(6,1) = y(7,1)
          x(6,2) = y(7,2)
          x(7,1) = y(4,1)
          x(7,2) = y(4,2)
          x(8,1) = y(9,1)
          x(8,2) = y(9,2)
          x(9,1) = y(1,1)
          x(9,2) = y(1,2)
          x(10,1) = y(6,1)
          x(10,2) = y(6,2)
          y(1,1) = x(1,1)
          y(1,2) = x(1,2)
          y(2,1) = x(2,1)
          y(2,2) = x(2,2)
          y(3,1) = x(3,1)
          y(3,2) = x(3,2)
          y(4,1) = x(4,1)
          y(4,2) = x(4,2)
          y(5,1) = x(5,1)
          y(5,2) = x(5,2)
          y(6,1) = x(6,1)
          y(6,2) = x(6,2)
          y(7,1) = x(7,1)
          y(7,2) = x(7,2)
          y(8,1) = x(8,1)
          y(8,2) = x(8,2)
          y(9,1) = x(9,1)
          y(9,2) = x(9,2)
          y(10,1) = x(10,1)
          y(10,2) = x(10,2)
          repeat=repeat+1
          GOTO 1
      END IF
      CALL TIMER(end)
      print*,'Run time (n=10):',end-start
      END

      SUBROUTINE S2 (y)
      REAL F1, F2
c specialized version of procedure Fftc2
      REAL y
      DIMENSION y(100,2)
      REAL oddv, evenv, term2
      DIMENSION oddv(2), evenv(2), term2(2)
      oddv(1) = y(6,1)
      oddv(2) = y(6,2)
      evenv(1) = y(1,1)
      evenv(2) = y(1,2)
      term2(1) = F1(oddv)
      term2(2) = F2(oddv)
      y(6,1) = evenv(1) - term2(1)
      y(6,2) = evenv(2) - term2(2)
      y(1,1) = evenv(1) + term2(1)
      y(1,2) = evenv(2) + term2(2)
      oddv(1  ) = y( 7 , 1 )
      oddv(2  ) = y( 7 , 2 )
      evenv(1) = y(2,1)
      evenv(2) = y(2,2)
      term2(1) = F1(oddv)
      term2(2) = F2(oddv)
      y(7,1) = evenv(1) - term2(1)
      y(7,2) = evenv(2) - term2(2)
      y(2,1) = evenv(1) + term2(1)
      y(2,2) = evenv(2) + term2(2)
      oddv(1) = y(8,1)
      oddv(2) = y(8,2)
      evenv(1) = y(3,1)
      evenv(2) = y(3,2)
      term2(1) = F1(oddv)
      term2(2) = F2(oddv)
      y(8,1) = evenv(1) - term2(1)
      y(8,2) = evenv(2) - term2(2)
```

```
      y(3,1) = evenv(1) + term2(1)
      y(3,2) = evenv(2) + term2(2)
      oddv(1) = y(9,1)
      oddv(2) = y(9,2)
      evenv(1) = y(4,1)
      evenv(2) = y(4,2)
      term2(1) = F1(oddv)
      term2(2) = F2(oddv)
      y(9,1) = evenv(1) - term2(1)
      y(9,2) = evenv(2) - term2(2)
      y(4,1) = evenv(1) + term2(1)
      y(4,2) = evenv(2) + term2(2)
      oddv(1) = y(10,1)
      oddv(2) = y(10,2)
      evenv(1) = y(5,1)
      evenv(2) = y(5,2)
      term2(1) = F1(oddv)
      term2(2) = F2(oddv)
      y(10,1) = evenv(1) - term2(1))
      y(10,2) = evenv(2) - term2(2))
      y(5,1) = evenv(1) + term2(1))
      y(5,2) = evenv(2) + term2(2))
      END

      REAL FUNCTION F1 (num2)
c specialized version of function Rcmul
      REAL num2
      DIMENSION num2(2)
      REAL hlp1
      hlp1 = 1.0 * num2(1)
      F1 = 0.0 * num2(2)
      F1 = hlp1 - F1
      RETURN
      END

      REAL FUNCTION F2 (num2)
c specialized version of function Icmul
      REAL num2
      DIMENSION num2(2)
      REAL hlp1
      hlp1 = 1.0 * num2(2)
      F2 = 0.0 * num2(1)
      F2 = hlp1 + F2
      RETURN
      END

      SUBROUTINE S3 (y)
      REAL F3, F4, F5, F6
c specialized version of procedure Fftc2
      REAL y
      DIMENSION y(100,2)
      REAL oddv, evenv, term2
      DIMENSION oddv(2), evenv(2), term2(2)
      oddv(1) = y(3,1)
      oddv(2) = y(3,2)
      evenv(1) = y(1,1)
      evenv(2) = y(1,2)
      term2(1) = F3(oddv)
      term2(2) = F4(oddv)
      y(3,1) = evenv(1) - term2(1)
      y(3,2) = evenv(2) - term2(2)
      y(1,1) = evenv(1) + term2(1)
      y(1,2) = evenv(2) + term2(2)
      oddv(1) = y(4,1)
      oddv(2) = y(4,2)
      evenv(1) = y(2,1)
      evenv(2) = y(2,2)
      term2(1) = F3(oddv)
      term2(2) = F4(oddv)
      y(4,1) = evenv(1) - term2(1)
      y(4,2) = evenv(2) - term2(2)
      y(2,1) = evenv(1) + term2(1)
      y(2,2) = evenv(2) + term2(2)
      oddv(1) = y(7,1)
      oddv(2) = y(7,2)
      evenv(1) = y(5,1)
      evenv(2) = y(5,2)
      term2(1) = F5(oddv)
      term2(2) = F6(oddv)
      y(7,1) = evenv(1) - term2(1)
      y(7,2) = evenv(2) - term2(2)
      y(5,1) = evenv(1) + term2(1)
      y(5,2) = evenv(2) + term2(2)
      oddv(1) = y(8,1)
      oddv(2) = y(8,2)
      evenv(1) = y(6,1)
      evenv(2) = y(6,2)
      term2(1) = F5(oddv)
      term2(2) = F6(oddv)
      y(8,1) = evenv(1) - term2(1)
      y(8,2) = evenv(2) - term2(2)
      y(6,1) = evenv(1) + term2(1)
      y(6,2) = evenv(2) + term2(2)
      END

      REAL FUNCTION F3 (num2)
c specialized version of function Rcmul
      REAL num2
      DIMENSION num2(2)
      REAL hlp1
      hlp1 = 1.0 * num2(1)
      F3 = 0.0 * num2(2)
      F3 = hlp1 - F3
      RETURN
      END

      REAL FUNCTION F4 (num2)
c specialized version of function Icmul
      REAL num2
      DIMENSION num2(2)
      REAL hlp1
      hlp1 = 1.0 * num2(2)
      F4 = 0.0 * num2(1)
      F4 = hlp1 + F4
      RETURN
      END

      REAL FUNCTION F5 (num2)
c specialized version of function Rcmul
      REAL num2
      DIMENSION num2(2)
      REAL hlp1
      hlp1 = 0.796274E-03 * num2(1)
      F5 = 1.0 * num2(2)
      F5 = hlp1 - F5
      RETURN
      END

      REAL FUNCTION F6 (num2)
c specialized version of function Icmul
      REAL num2
      DIMENSION num2(2)
      REAL hlp1
      hlp1 = 0.796274E-03 * num2(2)
      F6 = 1.0 * num2(1)
      F6 = hlp1 + F6
      RETURN
      END

      SUBROUTINE S4 (y)
      REAL F7, F8, F9, F10, F11, F12
      REAL F13, F14
c specialized version of procedure Fftc2
      REAL y
      DIMENSION y(100,2)
      REAL oddv, evenv, term2
      DIMENSION oddv(2), evenv(2), term2(2)
      REAL hlp
      oddv(1) = y(2,1)
      oddv(2) = y(2,2)
      evenv(1) = y(1,1)
      evenv(2) = y(1,2)
      term2(1) = F7(oddv)
      term2(2) = F8(oddv)
      y(2,1) = evenv(1) - term2(1)
      y(2,2) = evenv(2) - term2(2)
```

```
      y(1,1) = evenv(1) + term2(1)
      y(1,2) = evenv(2) + term2(2)
      oddv(1) = y(4,1)
      oddv(2) = y(4,2)
      evenv(1) = y(3,1)
      evenv(2) = y(3,2)
      term2(1) = F9( oddv)
      term2(2) = F10( oddv)
      y(4,1) = evenv(1) - term2(1)
      y(4,2) = evenv(2) - term2(2)
      y(3,1) = evenv(1) + term2(1)
      y(3,2) = evenv(2) + term2(2)
      oddv(1) = y(6,1)
      oddv(2) = y(6,2)
      evenv(1) = y(5,1)
      evenv(2) = y(5,2)
      term2(1) = F11(oddv)
      term2(2) = F12(oddv)
      y(6,1) = evenv(1) - term2(1)
      y(6,2) = evenv(2) - term2(2)
      y(5,1) = evenv(1) + term2(1)
      y(5,2) = evenv(2) + term2(2)
      oddv(1) = y(8,1)
      oddv(2) = y(8,2)
      evenv(1) = y(7,1)
      evenv(2) = y(7,2)
      term2(1) = F13(oddv)
      term2(2) = F14(oddv)
      y(8,1) = evenv(1) - term2(1)
      y(8,2) = evenv(2) - term2(2)
      y(7,1) = evenv(1) + term2(1)
      y(7,2) = evenv(2) + term2(2)
      END

      REAL FUNCTION F7 (num2)
c specialized version of function Rcmul
      REAL num2
      DIMENSION num2(2)
      REAL hlp1
      hlp1 = 1.0 * num2(1)
      F7 = 0.0 * num2(2)
      F7 = hlp1 - F7
      RETURN
      END

      REAL FUNCTION F8 (num2)
c specialized version of function Icmul
      REAL num2
      DIMENSION num2(2)
      REAL hlp1
      hlp1 = 1.0 * num2(2)
      F8 = 0.0 * num2(1)
      F8 = hlp1 + F8
      RETURN
      END

      REAL FUNCTION F9 (num2)
c specialized version of function Rcmul
      REAL num2
      DIMENSION num2(2)
      REAL hlp1
      hlp1 = 0.796274E-03 * num2(1)
      F9 = 1.0 * num2 ( 2 )
      F9 = hlp1 - F9
      RETURN
      END

      REAL FUNCTION F10 (num2)
c specialized version of function Icmul
      REAL num2
      DIMENSION num2(2)
      REAL hlp1
      hlp1 = 0.796274E-03 * num2(2)
      F10 = 1.0 * num2(1)
      F10 = hlp1 + F10
      RETURN
      END

      REAL FUNCTION F11 (num2)
c specialized version of function Rcmul
      REAL num2
      DIMENSION num2(2)
      REAL hlp1
      hlp1= 0.707388 * num2(1)
      F11 = 0.706825 * num2(2)
      F11 = hlp1 - F11
      RETURN
      END

      REAL FUNCTION F12 (num2)
c specialized version of function Icmul
      REAL num2
      DIMENSION num2(2)
      REAL hlp1
      hlp1 = 0.707388 * num2(2)
      F12 = 0.706825 * num2(1)
      F12 = hlp1 + F12
      RETURN
      END

      REAL FUNCTION F13 (num2)
c specialized version of function Rcmul
      REAL num2
      DIMENSION num2(2)
      REAL hlp1
      hlp1 = (-0.706262) * num2(1)
      F13 = 0.707951 * num2(2)
      F13 = hlp1 - F13
      RETURN
      END

      REAL FUNCTION F14 (num2)
c specialized version of function Icmul
      REAL num2
      DIMENSION num2(2)
      REAL hlp1
      hlp1 = (-0.706262) * num2(2)
      F14 = 0.707951 * num2(1)
      F14 = hlp1 + F14
      RETURN
      END
```

Appendix B
The Cubic Splines Interpolation

B.1 Source Program

```
c Content    : Cubic Splines Interpolation
c Author     : Romana Baier

c global variables
      COMMON sy
      REAL sy
      DIMENSION sy(100)
      COMMON syy
      REAL syy
      DIMENSION syy(100)
c local variables
      INTEGER a, n, b, i, type
      REAL h, t, a0, a1, a2, a3, d1, d2
      REAL z, factor, hlp, g
      REAL x, y, k, l, m, v
      DIMENSION x(100), y(100), k(100)
      DIMENSION l(100), m(100), v(100)
      INTEGER repeat, start, end

      CALL TIMER(start)
      repeat=1
1     IF (repeat .NE. 1000) THEN
c x( ) indicates the x-coordinates (of the measured values)
c x( ) is dynamic
c initialize x( ) with: 1,2.3,3,5,13,18,20
          x(1)=1
          . . .
          x(7)=20
c y( ) indicates the y-coordinates (of the measured values)
```

```
c y( ) is dynamic
c initialize y( ) with: 2,3,4,5,7,6,3
        y(1)=2
        . . .
        y(7)=3
c z indicates the x-coordinate to be computed; dynamic
        z=4
c n indicates the number of measured values; static
        n=7
c h indicates the distance between each two consecutive
c measured values; static
        h=1
c type indicates the type of range conditions; static
c type=1: natural conditions
c type=2: periodical conditions
c type=3: deBoor conditions
        type=2
c Determine the derivatives (sy)
    IF (type .EQ. 1) THEN
c natural range conditions
        DO 200 i=2,n-2
            k(i)=4/h
            l(i)=1/h
            m(i)=1/h
            d1=y(i)-y(i-1)
            d1=d1/(h*h)
            d2=y(i+1)-y(i)
            d2=d2/(h*h)
            sy(i)=3*(d2+d1)
200     CONTINUE
        k(1)=2/h
        l(1)=1/h
        m(1)=1/h
        k(n-1)=4/h
        l(n-1)=1/h
        m(n-1)=1/h
        k(n)=2/h
        d1=y(2)-y(1)
        d1=d1/(h*h)
        sy(1)=3*d1
        d1=y(n-1)-y(n-2)
        d1=d1/(h*h)
        d2=y(n)-y(n-1)
        d2=d2/(h*h)
        sy(n-1)=3*(d1+d2)
        sy(n)=3*d2
        CALL Tridia (n,k,l,m)
    ELSE
    IF (type .EQ. 2) THEN
c periodical range conditions
        DO 400 i=2,n-2
            k(i)=4/h
            l(i)=1/h
            m(i)=1/h
            d1=y(i)-y(i-1)
            d1=d1/(h*h)
            d2=y(i+1)-y(i)
            d2=d2/(h*h)
            sy(i)=3*(d2+d1)
400     CONTINUE
        k(1)=3/h
        l(1)=1/h
        m(1)=1/h
        k(n-1)=3/h
        d1=y(2)-y(1)
        d1=d1/(h*h)
        d2=y(n)-y(n-1)
        d2=d2/(h*h)
        sy(1)=3*(d1+d2)
        d1=y(n-1)-y(n-2)
        d1=d1/(h*h)
        d2=y(n)-y(n-1)
        d2=d2/(h*h)
        sy(n-1)=3*(d1+d2)
        CALL Tridia (n-1,k,l,m)
        DO 500 i=2,n-2
            v(i)=sy(i)
            syy(i)=0
500     CONTINUE
        v(1)=sy(1)
        syy(1)=1
        v(n-1)=sy(n-1)
        syy(n-1)=1
c This invocation of Tridia is eliminated during specialization
c to periodical range conditions. In order to achieve this
c elimination a static copy of Tridia (i.e. Tridia1) is used.
c The array syy( ) is static, hence Tridia1 becomes static.
        CALL Tridia1 (n-1,k,l,m)
        hlp=syy(1)+syy(n-1)+h
        factor=v(1)+v(n-1)
        factor=factor/hlp
        DO 600 i=1,n-1
            sy(i)=v(i)-factor*syy(i)
600     CONTINUE
        sy(n)=sy(1)
    ELSE
    IF (type .EQ. 3) THEN
c deBoor range conditions
        DO 300 i=2,n-2
            k(i)=4/h
            l(i)=1/h
            m(i)=1/h
            d1=y(i)-y(i-1)
            d1=d1/(h*h)
            d2=y(i+1)-y(i)
            d2=d2/(h*h)
            sy(i)=3*(d2+d1)
300     CONTINUE
        k(1)=1/h
        l(1)=2/h
        m(1)=1/h
        k(n-1)=4/h
        l(n-1)=1/h
        m(n-1)=2/h
        k(n)=1/h
        d1=y(2)-y(1)
        d1=d1/(h*h)
        d2=y(3)-y(2)
        d2=d2/(h*h)
        sy(1)=2*d1+0.5*(d1+d2)
        d1=y(n-1)-y(n-2)
        d1=d1/(h*h)
        d2=y(n)-y(n-1)
        d2=d2/(h*h)
        sy(n-1)=3*(d1+d2)
        sy(n)=2*d2+0.5*(d2+d1)
        CALL Tridia (n,k,l,m)
    END IF
    END IF
    END IF
    a=1
    b=n
700 i=(a+b)/2
    IF (x(i) .LT. z) THEN
        a=i
    ELSE
        b=i
    END IF
    IF ((a+1) .NE. b) THEN
        GOTO 700
    END IF
    i=a
    t=(z-x(i))/h
    a0=y(i)
    a1=y(i+1)-a0
    a2=a1-h*sy(i)
    a3=h*sy(i+1)-a1
    a3=a3-a2
    g=a0+(a1+(a2+a3*t)*(t-1))*t

    repeat=repeat+1
    GOTO 1
    END IF
```

```
c Return the value of the y-coordinate at the desired x-coordinate
      PRINT*,'g(z):',g
      CALL TIMER(end)
      print*,'Run time (period.):',end-start
      END

      SUBROUTINE Tridia (n,a,b,c)
c procedure Tridia solves tridiagonal equations
      COMMON sy
      REAL sy
      DIMENSION sy(100)
      INTEGER n
      REAL a, b, c
      DIMENSION a(100), b(100), c(100)
      REAL co, si, h, t, i, hlp
      b(n)=0
      DO 800 i=1,n-1
          IF (c(i) .NE. 0) THEN
              t=a(i)/c(i)
              si=1/SQRT(1+t*t)
              co=t*si
              a(i)=a(i)*co+c(i)*si
              h=b(i)
              b(i)=h*co+a(i+1)*si
              a(i+1)=(-h)*si+a(i+1)*co
              c(i)=b(i+1)*si
              b(i+1)=b(i+1)*co
              hlp=sy(i)
              sy(i)=hlp*co+sy(i+1)*si
              sy(i+1)=(-hlp)*si+sy(i+1)*co
          END IF
800   CONTINUE
      sy(n)=sy(n)/a(n)
      hlp=b(n-1)*sy(n)
      sy(n-1)=(sy(n-1)-hlp)/a(n-1)
      DO 900 i=n-2,1,-1
          hlp=b(i)*sy(i+1)-c(i)*sy(i+2)
          sy(i)=(sy(i)-hlp)/a(i)
900   CONTINUE
      END

      SUBROUTINE Tridia1 (n,a,b,c)
c Tridia1 is a static copy of Tridia
c procedure Tridia solves tridiagonal equations
      COMMON syy
      REAL syy
      DIMENSION syy(100)
      INTEGER n
      REAL a, b, c
      DIMENSION a(100), b(100), c(100)
      REAL co, si, h, t, i, hlp
      b(n)=0
      DO 801 i=1,n-1
          IF (c(i) .NE. 0) THEN
              t=a(i)/c(i)
              si=1/SQRT(1+t*t)
              co=t*si
              a(i)=a(i)*co+c(i)*si
              h=b(i)
              b(i)=h*co+a(i+1)*si
              a(i+1)=(-h)*si+a(i+1)*co
              c(i)=b(i+1)*si
              b(i+1)=b(i+1)*co
              hlp=syy(i)
              syy(i)=hlp*co+syy(i+1)*si
              syy(i+1)=(-hlp)*si+syy(i+1)*co
          END IF
801   CONTINUE
      syy(n)=syy(n)/a(n)
      hlp=b(n-1)*syy(n)
      syy(n-1)=(syy(n-1)-hlp)/a(n-1)
      DO 901 i=n-2,1,-1
          hlp=b(i)*syy(i+1)-c(i)*syy(i+2)
          syy(i)=(syy(i)-hlp)/a(i)
901   CONTINUE
      END
```

B.2 Residual Program

```
      COMMON sy
      REAL sy
      DIMENSION sy(100)
c specialized version of procedure Main
      REAL t, a0, a1, a2, a3, d1, d2
      REAL z, factor, g
      REAL x, y, v
      DIMENSION x(100), y(100), v(100)
      INTEGER repeat, start, end

      CALL TIMER(start)
      repeat=1
1     IF (repeat .NE. 1000) THEN
          . . .
c Initialize the x- and y-coordinates of the
c measured values as in the source program
          . . .
c z indicates the x-coordinate to be computed
          z = 4
          d1 = y(2) - y(1)
          d1 = d1 / 1
          d2 = y(3) - y(2)
          d2 = d2 / 1
          sy(2) = 3 * (d2 + d1)
          d1 = y(3) - y(2)
          d1 = d1 / 1
          d2 = y(4) - y(3)
          d2 = d2 / 1
          sy(3) = 3 * (d2 + d1)
          d1 = y(4) - y(3)
          d1 = d1 / 1
          d2 = y(5) - y(4)
          d2 = d2 / 1
          sy(4) = 3 * (d2 + d1)
          d1 = y(5) - y(4)
          d1 = d1 / 1
          d2 = y(6) - y(5)
          d2 = d2 / 1
          sy(5) = 3 * (d2 + d1)
          d1 = y(2) - y(1)
          d1 = d1 / 1
          d2 = y(7) - y(6)
          d2 = d2 / 1
          sy(1) = 3 * (d1 + d2)
          d1 = y(6) - y(5)
          d1 = d1 / 1
          d2 = y(7) - y(6)
          d2 = d2 / 1
          sy(6) = 3 * (d1 + d2)
c Procedure Tridia is called
          CALL S2 ( )
          v(2) = sy(2)
          v(3) = sy(3)
          v(4) = sy(4)
          v(5) = sy(5)
          v(1) = sy(1)
          v(6) = sy(6)
c the invocation of the procedure Tridia1 was performed by
c the interpreter at compile time
      factor = v(1)+v(6)
      factor = factor/1.65743
      sy(1) = v(1)-(factor*0.277513)
      sy(2) = v(2)-(factor*0.371867E-01)
      sy(3) = v(3)-(factor*(-0.886352E-01))
      sy(4) = v(4)-(factor*0.130002)
      sy(5) = v(5)-(factor*(-0.183686))
      sy(6) = v(6)-(factor*0.379919)
      sy(7) = sy(1)
      IF (x(4) .LT. z) THEN
          GOTO 63
      ELSE
          GOTO 64
      END IF
63    IF (x(5) .LT. z) THEN
          GOTO 65
```

```
      ELSE
         GOTO 66
      END IF
64    IF (x(2) .LT. z) THEN
         GOTO 74
      ELSE
         GOTO 75
      END IF
65    IF (x(6) .LT. z) THEN
         GOTO 83
      ELSE
         GOTO 91
      END IF
66    t = (z - x(4)) / 1
      a0 = y(4)
      a1 = y(5) - a0
      a2 = a1 - (1 * sy(4))
      a3 = (1 * sy(5))- a1
      a3 = a3 - a2
      g = a0+((a1+((a2+(a3*t))*(t-1)))*t)
      GOTO 114
74    IF (x(3) .LT. z) THEN
         GOTO 99
      ELSE
         GOTO 107
      END IF
75    t = (z - x(1)) / 1
      a0 = y(1)
      a1 = y(2) - a0
      a2 = a1 - (1 * sy(1))
      a3 = (1 * sy(2)) - a1
      a3 = a3 - a2
      g = a0+((a1+((a2+(a3*t))*(t-1)))*t)
      GOTO 114
83    t = (z - x(6)) / 1
      a0 = y(6)
      a1 = y(7) - a0
      a2 = a1 - (1 * sy(6))
      a3 = (1 * sy(7)) - a1
      a3 = a3 - a2
      g = a0+((a1+((a2+(a3*t))*(t-1)))*t)
      GOTO 114
91    t = (z - x(5)) / 1
      a0 = y(5)
      a1 = (y(6) - a0
      a2 = a1 - (1 * sy(5))
      a3 = (1 * sy(6))- a1
      a3 = a3 - a2
      g = a0+((a1+((a2+(a3*t))*(t-1)))*t)
      GOTO 114
99    t = (z - x(3)) / 1
      a0 = y(3)
      a1 = y(4) - a0
      a2 = a1 - (1 * sy(3))
      a3 = (1 * sy(4))- a1
      a3 = a3 - a2
      g = a0+((a1+((a2+(a3*t))*(t-1)))*t)
      GOTO 114
107   t = (z - x(2)) / 1
      a0 = y(2)
      a1 = y(3) - a0
      a2 = a1 - (1 * sy(2))
      a3 = (1 * sy(3))- a1
      a3 = a3 - a2
      g = a0+((a1+((a2+(a3*t))*(t-1)))*t)

114   repeat=repeat+1
      GOTO 1
      END IF
      PRINT*,'g(z) : ',g
      CALL TIMER(end)
      print*,'Run time (period.):',end-start
      END
```

```
      SUBROUTINE S2 ( )
c specialized version of procedure Tridia
      COMMON sy
      REAL sy
      DIMENSION sy(100)
      REAL hlp
      hlp=sy(1)
      sy(1)=(hlp*0.948683)+(sy(2)*0.316228)
      sy(2)=((-hlp)*0.316228)+(sy(2)*0.948683)
      hlp=sy(2)
      sy(2)=(hlp*0.961074)+(sy(3)*0.276289)
      sy(3)=((-hlp)*0.276289)+(sy(3)*0.961074)
      hlp=sy(3)
      sy(3)=(hlp*0.963174)+(sy(4)*0.268879)
      sy(4)=((-hlp)*0.268879)+(sy(4)*0.963174)
      hlp=sy(4)
      sy(4)=(hlp*0.963408)+(sy(5)*0.268039)
      sy(5)=((-hlp)*0.268039)+(sy(5)*0.963408)
      hlp=sy(5)
      sy(5)=(hlp*0.963431)+(sy(6)*0.267957)
      sy(6)=((-hlp)*0.267957)+(sy(6)*0.963431)
      sy(6)=sy(6)/2.63214
      hlp=1.73205*sy(6)
      sy(5)=(sy(5)-hlp)/3.73194
      hlp=(2.00009*sy(5))-(0.268039*sy(6))
      sy(4)=(sy(4)-hlp)/3.73080
      hlp=(2.00120*sy(4))-(0.268879*sy(5))
      sy(3)=(sy(3)-hlp)/3.71915
      hlp=(2.01691*sy(3))-(0.276289*sy(4))
      sy(2)=(sy(2)-hlp)/3.61939
      hlp=(2.21359*sy(2))-(0.316228*sy(3))
      sy(1)=(sy(1)-hlp)/3.16228
      END
```