Faster Fourier Transforms via Automatic Program Specialization

Julia L. Lawall

IRISA - Compose group Campus Universitaire de Beaulieu 35042 Rennes Cedex, France e-mail: jll@irisa.fr

Because of its wide applicability, many efficient implementations of the Fast Fourier Transform have been developed. We propose that an efficient implementation can be produced automatically and reliably by partial evaluation. Partial evaluation of an unoptimized implementation produces a speedup of over 9 times. The automatically generated result of partial evaluation has performance comparable to or exceeding that produced by a variety of hand optimizations. We analyze the benefits of partial evaluation at both compile time and run time, focusing on compiler issues that affect the performance of the specialized program.

1 Introduction

The Fourier transform and its inverse are widely used in a variety of scientific applications, such as audio and image processing [21], integration [21], and calculation using very large numbers [3,22]. The transform converts a function defined in terms of time to a function defined in terms of frequency. When a function is defined over the frequency domain, some expensive calculations are more tractable. This technique was made practical by the development of the Fast Fourier Transform (FFT) [7], which uses a divide-and-conquer algorithm to calculate the Fourier transform of a function represented as a discrete set of evenly-spaced data points. The divide-and-conquer algorithm reduces the complexity from $O(n^2)$ to $O(n\log n)$, where n is the number of data points.

Despite the significantly improved performance, the FFT remains an expensive operation. Many computations spend a substantial amount of time performing FFT's. For example, the 125.turb3d program of the SPEC95 Benchmark suite [8] spends about 40% of the time performing FFT's of 32 or 64 elements, using a hand-optimized implementation. Much effort has gone into hand-optimizing implementations of the algorithm. These optimizations include using recurrences to limit the number of calls to sine and cosine, eliminating the calls to these math library routines completely by reimplementing them more efficiently or using tables, reducing the number of real multiplications, and unrolling loops.

In this paper, we investigate whether partial evaluation is a suitable tool for generating an efficient FFT implementation. One measure of success is how many expressions are eliminated or simplified by partial evaluation. In this paper, we take a lower-level approach, and analyze the performance obtained on a

particular architecture (the Sun Ultrasparc). We find that partial evaluation improves the unoptimized implementation over 9 times when the input contains 16 elements and over 3 times when the input contains 512 elements. In an expanded version of this paper [16], we demonstrate that these results are competitive with the performance of hand optimization techniques, as illustrated by a variety of existing, publicly-available implementations.

The rest of this paper is organized as follows: Section 2 presents an overview of partial evaluation. Section 3 assesses the opportunities for specialization presented by the FFT algorithm, and estimates the speedup that can be obtained. Section 4 carries out the specialization of a simple implementation of the FFT. In Sections 5 and 6, we slightly rewrite the source program to get better results from specialization at compile-time and run-time, respectively. Finally, Section 7 describes other related work and Section 8 concludes.

2 Overview of partial evaluation

Partial evaluation is an automatic program transformation that specializes a program with respect to part of its input. Expressions that depend only on the known input and on program constants are said to be *static*. These expressions can be evaluated during specialization. Other expressions are said to be *dynamic*. These expressions are reconstructed to form the residual program. An *offline* partial evaluator begins with a *binding-time analysis* phase that determines which constructs are static and which are dynamic. Constructs are annotated to be evaluated or reconstructed accordingly. Binding-time analysis is followed by *specialization*, which builds the specialized program following these annotations.

We use the *Tempo* partial evaluator [5,13] for the C programming language. Tempo is the only partial evaluator for C that provides specialization at both compile time and run time, based on a single, compile-time, binding-time analysis. This structure is illustrated in Figure 1.

Compile-time specialization maps source code into specialized source code, based on invariants supplied at compile time. The specialized program may subsequently be compiled by any compiler. Thus, this approach is not tied to a particular compiler or architecture.

Run-time specialization specializes a program based on invariants that are not available until run time. Run-time specialization directly produces object code for a particular architecture. To limit the run-time overhead, Tempo constructs the specialized code out of code fragments, known as *templates*, that are compiled at compile time [6], *i.e.*, before run time. These templates contain *holes* to represent static subexpressions, whose values are not available until specialization. Specialization at run time consists of evaluating the static expressions, copying the compiled templates, and filling the holes. Experiments have shown that the overhead for specialization at run time following this approach is small [18].

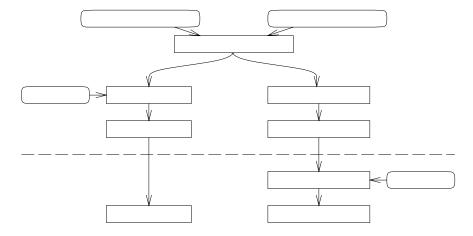


Fig. 1. Overview of Tempo

3 Why specialize the FFT?

We first consider why an implementation of the FFT is a good candidate for optimization via partial evaluation. The FFT is generally a module in another program. The benefit accrued from specializing a module depends on two factors: how often it is invoked with the inputs to which it is specialized, and what proportion of the computation of the module depends only on those inputs. The former requires examination of typical calling contexts, while the latter requires that we analyze the FFT implementation itself.

Using the analysis of the implementation, we then approximate the speedup obtainable from this approach. This approximation will be used to evaluate the success of our subsequent specializations of the FFT.

3.1 Typical calling contexts

We consider the FFT of a function represented as a one-dimensional array of complex numbers. In many applications, such an FFT is repeatedly applied to a stream of functions of fixed size. Furthermore, a multi-dimensional FFT can be implemented as a one-dimensional FFT iterated over all the rows and columns of the array. Thus, in a typical application, a one-dimensional FFT is applied many times to data sets of the same size. These observations suggest that it may be useful to specialize an implementation of the FFT to the number of elements in one dimension and the direction of the transformation.

3.2 Analysis of the implementation

An unoptimized implementation, adapted from Arndt [2], of the FFT is shown in Figure 2. The program consists of two procedures: fft and scramble. The arguments to the main procedure, fft, are two one-dimensional arrays (representing

the real and complex components of the function respectively), the size of each array, and an integer representing the direction of the transformation. Because of the divide-and-conquer strategy, each of the pair of arrays representing the function to transform has a size that is a power of two. The procedure fft first calls scramble to reorder the array elements, and then carries out the transform. The implementation follows the structure of the mathematical algorithm. This structure is typical of more optimized implementations.

The fonts used in Figure 2 represent the result of binding-time analysis with the array size and the direction of the transformation static. Static constructs are italicized, while dynamic constructs are in **bold** face.

The most computationally intensive part of the program consists of the three nested loops in the fft procedure. As indicated by the result of the binding-time analysis, this portion of the program contains the following opportunities for improvement by specialization:

- All loop tests are static. Thus loops can be unfolded, leading to straight-line code.
- The second of the three nested loops contains a static call to the sine function and a static call to the cosine function on each iteration. Eliminating these expensive function calls should lead to a significant performance improvement.
- In the innermost loop, there are numerous references to static variables within dynamic expressions. When specialization is performed at compile time, the compiler of the specialized code can exploit the resulting constant values. For example, the compiler can eliminate multiplications by 0 and 1.

Overall, both the opportunities for shifting significant computation to specialization time, and, in the case of compile-time specialization, the prospect of more effective compilation of the specialized program, make implementations of the FFT attractive candidates for partial evaluation.

3.3 Speedup estimation

While partial evaluation would seem intuitively to improve performance by eliminating expensive computations, it can also degrade performance by producing code that is too large to fit in the instruction cache, or to be compiled well. To assess the speedup obtained by partial evaluation, we characterize the expected speedup based on an analysis of the implementation. The expected speedup depends on two factors: the amount of code that is eliminated by specialization and the execution time of this code. We estimate the former using a complexity analysis of the implementation, described in terms of the number of elements n and the costs of the static and dynamic parts of the fft and scramble procedures. We estimate the latter using the speedup obtained by specialization to 16 elements (the smallest number of elements considered in our tests). Specialization for 16 elements produces a relatively small program, which we assume does not cause cache problems and can be compiled effectively. We then combine

```
#define M_PI 3.14159265358979323846
#define REAL double
#define SWAP(x, y) {REAL tmp = x; x = y; y = tmp; }
                                                   void
fft(REAL *fr, REAL *fi, int ldn, int is) scramble(REAL *fr, REAL *fi, int n)
   int\ n2, ldm, m, mh, j, r, t1, t2;
                                                      int\ m,j;
                                                      for (m=1, j=0; m< n-1; m++) {
   REAL pi, phi, c, s;
   REAL ur, vr, ui, vi;
                                                        for (k=n>>1; !((j^=k)@k); k>>=1);
   n2=1 << ldn;
  p i=is *M_PI;
                                                         if (j>m) \{
   scramble(fr,fi,n2);
                                                           SWAP(fr[m],fr[j]);
                                                           SWAP(fi[m],fi[j]);
   for (ldm=1; ldm < = ldn; ldm + +) {
      m = (1 << ldm);
                                                   }
      mh = (m >> 1);
     phi=pi/(REAL)(mh);
     for (j=0; j< mh; j++) {
        REAL \ w=phi*(REAL)j;
        c = cos(w);
        s = sin(w);
        for (r=0; r< n2; r+=m) {
           t1=r+j;
           t2=t1+mh;
           \mathbf{vr} = \mathbf{fr}[t2] * c - \mathbf{fi}[t2] * s;
           \mathbf{vi} = \mathbf{fr}[t2] * s + \mathbf{fi}[t2] * c;
           \mathbf{ur} = \mathbf{fr}[t1];
           \mathbf{fr}[t1] + = \mathbf{vr};
           \mathbf{fr}[t2] = \mathbf{ur} \cdot \mathbf{vr};
           ui=fi[t1];
           \mathbf{fi}[t1] + = \mathbf{vi};
           fi[t2]=ui-vi;
     }
  }
}
```

Fig. 2. Binding-time analysis of fft and scramble. Static expressions are in italics and dynamic expressions are in boldface.

this information to estimate the speedup expected for more values. N.B. For the fft and scramble procedures, the number of iterations of the inner loop(s) depends on the current value of the loop index of the outermost loop. To simplify the presentation, we describe the complexity of a loop as the number of times its body is executed per invocation of the enclosing procedure, rather than per invocation of the enclosing block of code.

The scramble procedure consists of two nested loops. Both loops are unrolled by specialization. The outer loop contains the completely static inner loop and some dynamic code. This dynamic code has complexity $\frac{n}{2}$. The static inner loop has complexity n. Let s_s represent the cost of one iteration of the static inner loop, and s_d represent the cost of one iteration of the dynamic part of the outer loop. Then, the complexity S_{un} of the unspecialized scramble procedure is:

$$S_{un} = ns_s + \frac{n}{2}s_d$$

The specialized procedure consists of just the dynamic code of the outer loop. Thus, its complexity S_{sp} is:

$$S_{sp} = \frac{n}{2} s_d$$

The fft procedure consists of three nested loops. All loops are unrolled by specialization. The complexity of the outermost loop is $\log n$. The complexity of the second loop is n. The complexity of the innermost loop is $\frac{n \log n}{2}$. The outer two loops consist of static code and the nested loop. Thus, only code from the body of the innermost loop appears in the residual program. The cost of the static code in the outermost loop should be negligible, and the complexity of the loop is comparatively small as well, so we omit the outermost loop from our complexity estimate. Let f_s represent the cost of the static code in the second loop, and f_d represent the cost of the dynamic code in the innermost loop. Then, the complexity F_{un} of the unspecialized fft procedure is:

$$F_{un} = nf_s + \frac{n\log n}{2}f_d$$

The complexity F_{sp} of the specialized procedure is:

$$F_{sp} = \frac{n \log n}{2} f_d$$

Solving the above equations for s_s , s_d , f_s , and f_d in terms of S_{un} , S_{sp} , F_{un} , and F_{sp} gives the following equations:

$$s_s = \frac{S_{un} - S_{sp}}{n} \quad f_s = \frac{F_{un} - F_{sp}}{n}$$

$$s_d = \frac{2}{n} S_{sp} \qquad f_d = \frac{2}{n \log n} F_{sp}$$

Substituting the actual execution times of the specialized and unspecialized procedures for 16 elements gives an estimate of the costs of the static and dynamic blocks of code.

The speedup obtained by specialization of the complete program is:

$$\frac{S_{un} + F_{un}}{S_{sp} + F_{sp}} = \frac{\frac{n}{2}s_d + ns_s + nf_s + \frac{n\log n}{2}f_d}{\frac{n}{2}s_d + \frac{n\log n}{2}f_d}$$

The expression on the right, instantiated with the estimated values of s_s , s_d , f_s , and f_d , calculates the expected speedup for any number of elements.

4 Specialization of a simple FFT implementation

We now assess the performance of the result of specializing the FFT implementation, shown in Figure 2, to the number of elements and the direction of the transformation static. We carry out specialization at both compile time and run time.

4.1 Methodology

The experimental results were obtained on a 200MHz Sun ultrasparc running Solaris (SunOS 5.5). The machine has 256MB of main memory, a 16KB primary data cache, a 16KB primary instruction cache, and a 512KB secondary cache. Programs were compiled with gcc version 2.8.1 using the options -02 -mcpu=ultrasparc -ffast-math and Sun's cc compiler version 4.2 using the options -fast -x05 -xinline=[]. These are the maximum optimization levels for these compilers, omitting inlining, which is not interesting for this program. For run-time specialization, templates were compiled with gcc with the additional option -fno-schedule-insns to prevent unwanted scheduling optimizations within templates. This option was not used in compiling the unspecialized reference program. Run times were calculated using getrusage and include only the user time.

4.2 Compile-time specialization

The specialized program consists of the specialized fft procedure, which calls the specialized scramble procedure. The specialized fft procedure is straightline code, built from the dynamic code in the innermost loop. The calls to sin and cos have been eliminated, and the uses of their values have been replaced by explicit constants. Array indices are constant as well. The specialized scramble procedure is similar.

Figure 3 shows the performance of the specializations for 16 to 512 elements. The expected speedups are shown in parentheses after the actual speedups, and are based on applying the formula derived in Section 3.3 to the speedup obtained

for 16 elements. For 32 and 64 elements, the actual speedup achieved when using gcc is slightly lower than the expected speedup, while the actual speedup achieved when using cc is slightly higher than the expected speedup. In both cases there is a significant drop off in performance at 128 elements. We examine some reasons for this behavior in Section 5.

	gcc			СС		
•	Source	Compile	time specialization	Source	Compile	time specialization
Size	$_{ m Time}$	$_{ m Time}$	${f Speedup}$	Time	$_{ m Time}$	${f Speedup}$
16	40.89	4.54	9.00	34.94	4.66	7.50
32	89.63	11.53	7.77 (8.09)	76.27	10.21	7.47 (6.90)
64	193.07	27.73	$6.96 \ (7.37)$	160.74	23.41	6.87 (6.40)
128	402.10	73.46	5.47 (6.78)	331.45	71.95	4.61 (5.98)
256	818.38	199.50	4.10 (6.29)	678.74	143.65	4.72 (5.62)
512	1682.42	518.32	$3.25\ (5.87)$	1398.50	337.74	4.14 (5.31)

Fig. 3. Performance of the source program and compile-time specializations (times in microseconds).

4.3 Run-time specialization

The structure of the code produced by run-time specialization is the same as the structure of the code produced by compile-time specialization. The speedups obtained by run-time specialization are shown in Figure 4. Again, the expected speedups, shown in parentheses after the actual speedups, are based on applying the formula derived in Section 3.3 to the speedup obtained for 16 elements. Figure 4 also shows the cost of specialization, itself. Because specialization is performed at run time, the specialized code must be run several times to pay for the cost of specialization. The number of runs required to amortize this cost is shown in the column labeled "=". (Where specialization slows down the program, this value is ∞ .)

The speedups obtained by run-time specialization are quite low, when compared to the speedups obtained by compile-time specialization. Furthermore, for more than 16 elements, the speedup is consistently lower than the expected speedup. For 512 elements, the specialized program is actually substantially slower than the original program. The time required to generate the specialized program is also substantial. We examine some solutions to these problems in Section 6.

5 Improving the result of compile-time specialization

Compile-time specialization of the FFT implementation produces two quite large procedures containing many integer and floating-point constants. These features

	Source	Run-time specialization					
Size	Time	Generate	$_{ m Time}$	${f Speedup}$	=		
16	40.74	366.59	12.07	3.38	13		
32	89.19	893.12	34.40	2.59 (2.96)	17		
64	188.53	2130.64	88.01	2.14 (2.67)	22		
128	390.71	4923.36	203.10	1.92 (2.45)	27		
256	812.21	11089.39	476.79	1.70 (2.29)	34		
512	1669.31	25691.15	2584.33	$0.65\ (2.15)$	∞		

Fig. 4. Performance of the source program and run-time specializations (gcc only, times in microseconds).

are atypical of handwritten C code, and thus hardware and compilers may not be optimized for such programs. We now examine the effects of cache behavior and compiler optimizations on the performance of specialized programs.

5.1 Hardware considerations

The ultrasparc has a 16KB primary instruction, a 16KB data cache, and a 512KB secondary cache. Figure 5 presents the size of the assembly code for the specialized programs and the size of the dynamic arrays. In all of our experiments the data fits within the primary data cache. For more than 64 elements, the specialized code does not fit within the primary instruction cache. Nevertheless, both the data cache and the instruction cache affect the performance of the specialized program.

	Array	Speciali	zed code size
Size	size	cc	gcc
16	0.256	2.036	2.052
32	0.512	5.168	5.188
64	1.024	12.860	12.868
128	2.048	30.228	32.612
256	4.096	70.132	82.852
512	8.192	158.948	164.004

Fig. 5. Data and code sizes (KB)

The behavior of the data cache has a significant effect on the performance of the specialized programs. In our experiments, we iterate the fft procedure many times on the same pair of arrays. In the best case, all of the array elements remain in the cache between each iteration. Thus there are few data cache misses during the execution of the fft procedure. In the worst case, all the array elements are overwritten with values in other locations between each iteration.

Figure 6 compares the performance at these two extremes. For readability, we normalize the runtime by the complexity of the implementation, following the strategy of Frigo and Johnson [10]. For both gcc and cc, the specialized program is significantly faster when the data cache is preserved between iterations. Overwriting the data cache between iterations of the fft procedure slows down the unspecialized program only 3-5%, however. We have thus omitted this case from the figure.

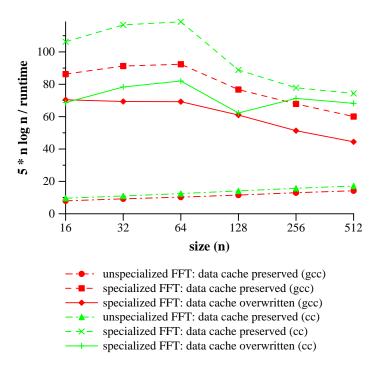


Fig. 6. Performance of the unspecialized and specialized implementations, comparing data cache behavior.

As shown in Figure 5, for more than 64 elements, the size of the specialized program exceeds the size of the instruction cache. Figure 6 shows that in all cases there is a drop off in performance at this point. The drop off is more significant when the data cache remains intact between iterations. When the specialized program is compiled with gcc the performance declines steadily, whereas when the specialized program is compiled with cc the performance levels off. Thus, we now turn to the difference between the behavior of cc and gcc on the specialized FFT implementation. We only consider the case where the data cache is overwritten between iterations, since a real application would likely apply the FFT to a different pair of arrays each time, or perform other calculations between successive calls to the FFT that would modify the data cache.

5.2 Compiler considerations

We now examine the effect of compiling the specialized FFT programs, focusing on the treatment of floating-point constants and the effect of very large procedures on register allocation.

Floating-point constants A reference to a small integer constant can be inlined in a machine instruction. A floating-point constant, however, is stored in the data segment of the machine code. Thus, loading such a constant may require memory access. In the compiled unspecialized program, the real values produced by the calls to the sine and cosine functions are stored in registers throughout the innermost loop. In the straight-line code produced by specialization, the extent of the reuse of a particular floating-point constant is less evident.

As shown in Figure 7, compilers vary in their treatment of the floating-point constants in the specialized FFT program. Version 2.8.1 of gcc consistently performs the fewest loads per floating-point constant. An earlier version of gcc, however, performed significantly more loads per floating-point constant for 64 or more elements. In contrast, version 4.2 of cc performs more loads per floating-point constant than an earlier version. Even when there are few loads per floating-point constant, the need to keep floating-point constants in registers prevents these registers from being used for other values.

Size	16	32	64	128	256	512
gcc 2.7.2.1 loads per float	1.0	1.0	3.5	6.4	9.7	13.4
gcc 2.8.1 loads per float	1.0	1.0	1.4	$^{2.3}$	3.1	3.5
cc 4.0 loads per float	1.0	1.0	1.0	1.4	2.1	3.0
cc 4.2 loads per float	1.0	1.2	2.2	$^{2.6}$	3.2	3.5

Fig. 7. Average loads per distinct floating-point constant.

Register allocation within very large procedures Procedure size affects the compiler's ability to allocate registers. For the source program, both cc and gcc generate assembly code that stores all intermediate values in registers. Ignoring for the moment the need to store floating-point constants in registers, it should be possible to compile the specialized program to use only registers as well.

When a compiler is unable to allocate registers to all the temporary values, it moves values from registers onto the stack. This operation is relatively expensive. In compiling the FFT program specialized for 128 or 256 elements, gcc generates many stack operations, including storing values on the stack that are never read again. No stack operations are generated by cc. These extra memory references account for some of the drop off in performance of the specialized program when compiled by gcc.

5.3 Obtaining better performance from the specialized code

The useless stack operations generated by gcc suggest that the complex operations performed at optimization level -02 perform badly on the large blocks of simple straight-line code. Compiling instead with optimization level -01, using the additional option -fschedule-insns to perform some instruction scheduling, both eliminates all the stack operations and produces faster code, as shown in Figure 8. Nevertheless, because of exceeding the instruction cache, we do not obtain the expected speedup for large numbers of elements.

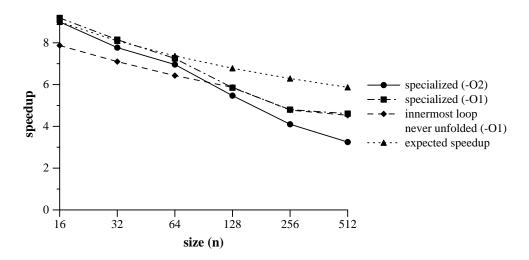


Fig. 8. Performance of variants of the specialized program (compiled with gcc).

At optimization level -01, gcc performs slightly more loads per floating-point constant than at optimization level -02. By avoiding optimizations that are harmful, we have also eliminated some optimizations that could be beneficial. Thus, particularly because specialized programs do not have the form of typical C programs, there are some trade-offs involved in obtaining the best performance.

Overall, the problems with compiling and running large procedures suggest that we may recover some of the expected speedup by performing less specialization, thus generating smaller procedures. A particularly appealing place to limit specialization is the innermost loop. Here the results of the expensive sine and cosine operations are used repeatedly, and the extra benefit of having static array indices seems minimal.

One approach is to simply never unroll the innermost loop. When gcc is used at optimization level -01 (Figure 8), the specialized rewritten program has essentially the same performance for larger numbers of elements as the specialized original program. Compiling with optimization level -02 gives similar performance. When cc is used (Figure 9), the rewritten specialized program has

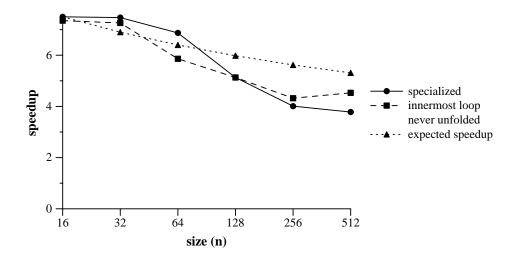


Fig. 9. Performance of variants of the specialized program (compiled with cc).

better performance for larger numbers than the specialized original program. Because of the instruction cache, however, the performance is again still lower than the expected speedup.

6 Improving the result of run-time specialization

Run-time specialization introduces different performance considerations. Because compilation is performed at compile time on a set of templates whose total size is bounded by the size of the source program, the duplication of templates at specialization time does not affect the quality of the compiled code. On the other hand, because the templates are compiled before the static constants become available, the compiler cannot take advantage of their values. By rewriting the program slightly, we can improve on the performance shown in Figure 4, and indeed achieve much of the speedup obtained by compile-time specialization. These transformations are described below.

6.1 The problem of a static value in a dynamic context

A static value in a dynamic context results in a *hole* within a template. Holes are represented such that each hole is considered by the compiler to be a unique, unknown constant [18]. The run-time specializer modifies the assembly code of the template by writing a load of the static value on top of the assembly code generated for the hole. This approach can harm the quality of the specialized code by preventing compile-time simplifications and by reducing opportunities for reuse of values loaded from memory.

To resolve these problems, we shift some specialization-time values to compile time (by explicitly enumerating the possible values), and shift others to execution time (by making them dynamic). Both approaches reduce the number of holes, which shortens specialization time and produces more efficient specialized code.

6.2 Compile-time optimizations based on explicit constants

Because static values are not available at compile time, run-time specialization does not introduce opportunities for compile-time optimizations based on explicit constants, such as eliminating multiplications by 0 or 1. To achieve this effect during run-time specialization, we introduce conditionals that test for these values and hand specialize the code accordingly. These extra conditionals are reduced during specialization, and thus only affect the specialization time. This rewriting is a variant of what is widely known within the partial evaluation community as "The Trick" [14].

For 16 elements the result of specialization after this optimization achieves a speedup of 4.25, which is significantly better than the speedup of 3.38 obtained by specialization of the original program. For more than 64 elements there is less improvement, reflecting the smaller percentage of computations affected by the optimization.

6.3 Sharing

The array indices of the innermost loop are also static values occurring in dynamic contexts. These values do not trigger any optimizations, so we simply instruct the binding-time analysis to consider these values as dynamic. This annotation strategy dramatically increases the amount of sharing in the specialized program.

In the innermost loop of the source program, several array elements are referenced repeatedly, without intervening writes. Thus the compiler can detect that the value stored in a register after the first load of such an array element can be used in the second occurrence, avoiding a second memory access. Run-time specialization, however, obscures this property completely. Each hole is encoded as a unique, unknown constant. Thus the compiler generates a separate memory access for each array reference. Making the array index dynamic reintroduces the possibility of sharing these values. The values of the calls to cosine and sine that are not eliminated by the test for 0 and 1 are also shared after this optimization. With this optimization, the speedup obtained ranges from 6.25 for 16 elements to 3.60 for 512 elements, in contrast to the speedups of between 3.38 and 0.65 obtained by the specialized original program.

6.4 Combining the optimizations

As shown in Figure 10, with both optimizations, the speedups range from 7.81 for 16 elements to 3.79 for 512 elements. Beginning with 64 elements, the actual

speedup is significantly lower than the expected speedup. In every case, the run-time specialized code is larger than the compile-time specialized code. The run-time specialized code exceeds the size of the instruction cache starting at 64, rather than 128, elements (c.f. Figure 5). Combining the two optimizations also reduces the generation time by a factor of two. The cost of specialization is amortized after at most 9 invocations.

The final column of Figure 10 compares the execution time of the run-time specialized code with the execution time of the result of compile-time specialization of the unoptimized program, compiled at optimization level -01. With both optimizations, run-time specialization achieves up to 76% of the performance of compile-time specialization. This figure is comparable to the results of other experiments with Tempo's run-time specialization [18].

-	Source	Run-time specialization				CT/RT
Size	$_{ m Time}$	${\rm Generate}$	$_{ m Time}$	$\mathbf{Speedup}$	=	Time
16	40.66	205.22	5.20	7.81	6	76%
32	89.90	451.71	12.90	6.97 (6.86)	6	76%
64	192.06	1020.68	35.76	5.37 (6.14)	7	66%
128	393.78	2279.70	94.32	4.17 (5.58)	8	64%
256	819.32	5045.29	204.78	4.00 (5.13)	9	74%
512	1678.38	10937.41	442.27	3.79 (4.76)	9	73%

Fig. 10. Performance of the result of run-time specialization after all optimizations to the source program (gcc only, times in microseconds).

7 Related work

Because we are investigating the application of partial evaluation to scientific code, we focus on partial evaluators for Fortran and C.

Glück et al. have specialized the FFT using their partial evaluator for Fortran [11]. They achieve speedups ranging from 5.05 for 16 elements to 1.83 for 512 elements. Their good results on this example motivated our investigation.

C-Mix is a partial evaluator for C developed by Andersen [1]. Using C-Mix, it should be possible to achieve similar results similar to our results for compile-time specialization. C-Mix does not provide run-time specialization. Furthermore, to achieve good results from run-time specialization, we require a binding-time analysis that can consider a variable to be both static and dynamic. C-Mix does not provide this facility.

Grant et al. have also developed a run-time specializer for C [12]. Like Tempo, their approach is based on templates. Their system allows the user to annotate particular program points as static or dynamic, rather than only allowing entry point annotations. This facility could be useful in our experiments to specify that a loop should not be unrolled or that a static variable in a dynamic context

should be treated as dynamic. Their system, however, provides fewer automatic analyses than Tempo, and thus relies more heavily on user annotations. Unlike Tempo, their system performs optimizations on the code produced by run-time specialization. These optimizations require extra run-time overhead. As shown by Figure 10, even without optimization at run-time, we obtain between 64% and 76% of the performance obtained by compile-time specialization, which can be viewed as specialization with maximal optimization of the specialized code. Their system does not provide compile-time specialization.

Tick C is a C-like language for describing code to be generated at run time, using Lisp-like backquote and comma operators [9]. Tick C is not a partial evaluator. Thus the code to generate the specialized program must be written by hand.

In this paper, we have assessed our experimental results by comparison with an expected speedup based on the complexity of the static and dynamic parts of the program. Jones *et al.* have also considered the problem of estimating the speedup achieved by partial evaluation [14]. Their analysis, however, considers only the limit of the speedup as the size of the input increases. In the case of the FFT, the limit is 1, indicating no speedup, because the dynamic code of the innermost loop of the fft procedure has a greater complexity than the static parts of the program.

8 Conclusion

In this paper, we have investigated the problem of generating an efficient FFT implementation by using automatic program specialization. We obtained significant performance improvement from both compile-time specialization (up to over 9 times faster) and run-time specialization (up to over 7 times faster). We carefully assessed compiler characteristics that affect the performance of specialized programs, and proposed simple rewritings of the source program to generate specialized programs that have better performance. Elsewhere, we have shown that this specialized FFT implementation, generated automatically and thus reliably, has performance comparable to hand optimized implementations [16].

While the partial evaluation techniques we have focused on, compile-time and run-time specialization, are not practical for FFT's of very large sequences, there are realistic applications, such as the 125.turb3d simulation program in the SPEC95 Benchmark suite [8], that repetitively perform FFT's of data sets of the sizes used in our experiments. Furthermore, data specialization [4, 15, 17] shows promise for extending automatic optimization within the partial-evaluation framework to much larger data sets.

A focus of this work has been to analyze the benefits of compile-time and run-time specialization given particular compilers and program rewritings before specialization. Two observations stand out. For compile-time specialization, the benefits of loop unrolling are offset by the possibility of exceeding the size of the instruction cache and the problems of compiling large procedures. Thus, specialization with respect to loop indices is not always beneficial. For run-time

specialization, we have seen a significant performance decline when there are many trivial holes. Considering a static variable in a dynamic context to be dynamic more than doubles the speedup obtained by run-time specialization of the FFT. These observations suggest that while compile-time and run-time specialization can share the same preprocessing framework, as illustrated in Figure 1, at the low level, different annotation strategies are appropriate. Further work is required to fully assess this approach. It is hoped that the analysis presented in this paper will prove useful in guiding the application of specialization, and in particular of Tempo, to other programs, and will motivate similar detailed analyses of the benefits of specialization.

Acknowledgements

The author would like to thank Gilles Muller, Olivier Danvy, Renaud Marlet, and Ulrik Schultz for helpful comments on the organization of the paper, André Seznec and Pascal Rigaux for help in understanding the Sparc architecture, as well as Charles Consel and the entire Compose group for support and encouragement during this work.

References

- L.O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.
- J. Arndt.
 URL: http://www.jjj.de/fxt/fxt970929.tgz in the file hfloat/src/fxt/simplfft/fft.c.
- 3. E. Bach and J. Shallit. Algorithmic Number Theory. The MIT Press, 1996.
- 4. G. J. Barzdins and M. A. Bulyonkov. Mixed computation and translation: Linearisation and decomposition of compilers. Preprint 791, Computing Centre of Siberian division of USSR Academy of Sciences, Novosibirsk, 1988.
- C. Consel, L. Hornof, F. Noël, 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.
- C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In POPL96 [20], pages 145-156.
- J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation, 19(90):297-301, April 1965.
- 8. Standard Performance Evaluation Corporation. SPEC95. URL: http://www.specbench.org.
- 9. D.R. Engler, W.C. Hsieh, and M.F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In POPL96 [20], pages 131–144.
- M. Frigo and S. Johnson. FFTW user's manual, 1997. URL: http://theory.lcs.mit.edu/~fftw/.

- 11. 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.
- B. Grant, M. Mock, M. Philipose, C. Chambers, and S.J. Eggers. Annotation-directed run-time specialization in C. In PEPM'97 [19], pages 163-178.
- 13. L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In PEPM'97 [19], pages 63-73.
- N.D. Jones, C. Gomard, and P. Sestoft. Partial Evaluation and Automatic Program Generation. International Series in Computer Science. Prentice-Hall, June 1993.
- 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.
- J.L. Lawall. Faster Fourier transforms via automatic program specialization. Publication interne 1192, IRISA, Rennes, France, May 1998.
- 17. K. Malmkjær. Program and data specialization: Principles, applications, and self-application. Master's thesis, DIKU, University of Copenhagen, Denmark, August 1989.
- 18. F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based runtime 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.
- 19. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997. ACM Press.
- 20. Conference Record of the 23^{rd} Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages, St. Petersburg Beach, FL, USA, January 1996. ACM Press.
- 21. W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C The Art of Scientific Computing*. Cambridge University Press, Cambridge, 2nd edition, 1995.
- 22. A. Schönhage, A.F.W. Grotefeld, and E. Vetter. Fast Algorithms: A Multitape Turing Machine Implementation. BI-Wissenschaftsverlag, 1994.