Using Program Specialization to Speed SystemC Fixed-Point Simulation

Stephen A. Edwards *

Columbia University, New York

sedwards@cs.columbia.edu

Abstract

Generic simulation components, such as fixed-precision arithmetic routines, make it easier to quickly assemble system simulations, but generic components tend to simulate more slowly than their manually-written specialized counterparts. So a system modeler is normally forced to choose between building a simulation quickly or running it quickly.

This paper explores the use of program specialization as a way to address this conundrum. Through hints provided by the author of a generic library and aggressive compiler optimizations, program specialization can automatically rewrite a generic component into a specialized one with performance comparable to a careful manual implementation. As a result, the user of such a specializable library can quickly assemble a simulation from generic components whose performance can equal that of a more tedious implementation.

Experimental results show that program specialization provides a three- to seven-times speed-up on an important class of simulations: signal processing kernels in SystemC that manipulate fixed-precision numbers.

Categories and Subject Descriptors D.3.4 [*Software*]: Programming Languages—Processors; I.6.3 [*Simulation and Modeling*]: Applications

General Terms Languages, Simulation

Keywords Program Specialization, Fixed-point Simulation, SystemC, Tempo, Prespec

1. Introduction

While great progress has been made in devising alternate ways to validate digital systems, simulation remains king because of its simplicity, scalability, and ability to provide a variety of insights into the behavior of a system. Its only problem? Designers always want it to be faster.

This paper presents a case study in using program specialization to speed a particular class of simulations: signal processing algorithms using fixed-point arithmetic coded in the popular SystemC

---

libraries [6]. Such arithmetic, which uses nonstandard word sizes and binary points in the middle of words, is easily implemented in hardware, but often time-consuming to simulate in software because most processors are best suited performing arithmetic on a handful of word sizes. Compounding the problem, especially in the case of the SystemC libraries, is the need to make the code general enough to work with many different bitwidths.

This paper makes two contributions: an experimental study that demonstrates the feasibility and effectiveness of using program specialization to accelerate simulation and a set of guidelines for writing libraries that can be effectively specialized. While the experiments conducted here were performed on signal-processing algorithms that use the SystemC fixed-point arithmetic libraries, the results should generalize to other simulation applications.

I experimented on signal-processing algorithms with fixed-point arithmetic because they are good candidates for specialization. Signal processing algorithms are usually written first using floating-point arithmetic to establish their viability, then refined to use fixed-precision arithmetic as a prelude to hardware implementation. Although some techniques for choosing word lengths are able to avoid simulation by working symbolically (e.g., Fang et al. [5]), most rely on bit-accurate simulation in the inner loop of an optimization algorithm, making fast simulation critical. Even when practical analytical techniques can be employed, efficient simulation code for a signal processing algorithm is still valuable because it may be needed in simulating the larger system that it is inevitably a part of.

Figure 1 shows a fixed-point multiplication routine. Its representation for a fixed-precision number—*struct fp*—consists of a machine word to hold the actual bits (the *val* field) along with configuration parameters such as *wl*, the word width, *iwl*, the integer portion of the word, etc. Figure 3 shows this representation graphically. For particular concrete values of these parameters, the specialized *mult* function in Figure 2 is equivalent, and naturally must faster, but a library needs to contain the more general code in Figure 1 because it must support arbitrary word sizes. The result, not surprisingly, is that typical signal-processing algorithms that use the library code can run ten to one hundred times slower than necessary. In this paper, I present experimental results that show program specialization is able to recover much of this, provided the code was written with specialization in mind. Specialization-ignorant library code only recovers a factor of two or three.

The specialized multiplication code in Figure 2 was produced automatically by the Tempo [3, 4] program specializer from the generic code in Figure 1 (I modified it only for clarity). Information about word length, etc. was provided by the Prespec [14] specification in Figure 4 and its effects are taken into account by the specializer. In particular, decisions derived from the word length, rounding, and quantization modes have been made at compile time and therefore do not consume time while the simulation is running.

```
typedef struct fp {
  int val; /* 32-bit value value of the number */
  int wl;  /* Word length, in bits */
  int iwl; /* Integer word length, in bits */
  int lbp; /* Location of binary point, in bits */
  int overflow;
  int rounding;
} fixed;

#define WL(f)  ((f)->wl)   /* Word length */
#define IWL(f) ((f)->iwl) /* Integer word length */
#define FWL(f) \
  ((f)->wl - (f)->iwl)    /* Fractional word length /*
#define LBP(f) ((f)->lbp) /* Location of binary point */
#define VAL(f) ((f)->val) /* Integer value */

#define POW2(n) (1 << (n)) /* 2^n for integers */

void mult(fixed *r, fixed *a, fixed *b) {
  int av, bv, shift;
  av = VAL(a) >> (LBP(a) - FWL(a));
  bv = VAL(b) >> (LBP(b) - FWL(b));
  shift = FWL(a) + FWL(b) - LBP(r);
  VAL(r) = av * bv;
  if (shift > 0) VAL(r) >>= shift;
  else if (shift < 0) VAL(r) <<= -shift;
  fix_quantize(r);
  fix_overflow(r);
}

void quantize(fixed *r) {
  int shift, delta, mask;
  switch (r->rounding) {
  case ROUND:
    delta = POW2(LBP(r) - FWL(r) - 1);
    shift = LBP(r) - FWL(r);
    VAL(r) =
      ((VAL(r) + delta) >> shift) << shift;
    break;
  case TRUNCATE:
    mask = POW2(LBP(r) - FWL(r)) - 1;
    VAL(r) &= ~mask;
    break;
  }
}

void overflow(fixed *r) {
  int min, max, shift, tmp;
  switch (r->overflow) {
  case SATURATE:
    max = POW2(IWL(r) + LBP(r) - 1) -
        POW2(LBP(r) - FWL(r));
    min = -POW2(IWL(r) + LBP(r) - 1) -
        POW2(LBP(r) - FWL(r)) - 1;
    if (VAL(r) > max) VAL(r) = max;
    else if (VAL(r) < min) VAL(r) = min;
    break;
  case WRAP:
    shift = sizeof(int) - IWL(r) - LBP(r);
    VAL(r) = (VAL(r) << shift) >> shift;
    break;
  }
}
```

Figure 1. Generic fixed-precision multiplication routines.

The usual approach to speeding the simulation of such fixed-point-arithmetic code is to employ an automatic code generator that synthesizes the specialized code from a higher-level description. The FRIDGE system from Keding et al. [10, 11] is representative and successful, but somewhat limited because each arithmetic algorithm and optimization must be hand-coded. The quality of the code is good—I found a program specializer can be coaxed into producing code with comparable performance—so the main advantage of program specialization is flexibility: new fixed-point arithmetic algorithms do not require source-level changes to the code generator.

```
void mult(fixed *r, fixed *a, fixed *b) {
  int av, bv;
  av = a->val >> 4;
  bv = b->val >> 4;
  r->val = av * bv;
  r->val >>= 8;
  r->val &= 0xfffffff0;
  if (r->val > 0x7fff0)
    r->val = 0x7fff0;
  else if (r->val < -0x80011)
    r->val = -0x80011;
}
```

Figure 2. The *mult* function of Figure 1 after specializing with wl=16, iwl=4, lbp=16, quantization=TRUNCATE, and overflow=SATURATE. Most computations have been done at compile time and functions have been inlined. This code runs about 2.5 times faster than the generic version in Figure 1.

In the remainder of this paper, I describe fixed-precision arithmetic, program specialization, and present experiments that show it can accelerate simulation as much as seven times. I conclude with a section on lessons for library writers who want to apply program specialization to their code.

2. Fixed-Precision Arithmetic

Arbitrary-precision arithmetic performed using limited-precision operations is a classical problem (see, for example, Knuth [12, section 4.3]). Algorithms for basic arithmetic operations are simple loops with a lot of bookkeeping that is usually specific only to the word length and not the values being manipulated. It is exactly this character that makes fixed-precision arithmetic a good candidate for program specialization.

The fixed-point libraries of SystemC [6] are the subject of the specialization experiments presented here. These libraries are written in C++.

Unfortunately, the fixed-point routines are not especially efficient. The experimental results in Table 1 show that they can be as much as three orders of magnitude slower than using native floating-point arithmetic. Carefully recoding certain routines without changing data representations can recover nearly a factor of seven (second row in Table 1), and program specialization can recover another factor of two.

Others have attempted to speed the simulation of signal processing algorithms that employ fixed-point arithmetic. The FRIDGE system from Keding et al. [10, 11] is representative and successful.

Flexibility is the main advantage of program specialization over FRIDGE. The fixed-precision arithmetic algorithms in FRIDGE are integrated in a code generator, making it difficult to modify or add algorithms. By contrast, a library developer using the program specialization technique used here does not need to write a code generation algorithm, just generic code for components written in a style that produces good results when program specialization is applied. In fact, the program specializer does not "understand" that the routines in Figure 1 perform multiplication; it does not need to.

2.1 Implementing Fixed-Precision Arithmetic

Figure 1 shows an implementation of a fixed-precision multiply algorithm that uses a contiguous sequence of bits in a machine word to represent a fractional number, shown in Figure 3. The *wl* parameter indicates the total number of bits used in the number representation, *iwl* is the number of bits to the left of the binary point (the integer word length), and *lbp* is location of the binary point in the machine word.
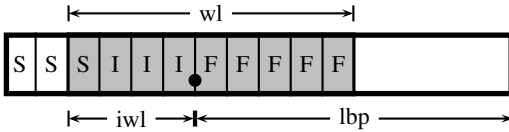
Figure 3. The representation of fixed-precision numbers used in the FRIDGE system [10], characterized by the parameters wl, iwl, and lbp.

Manipulating fixed-precision numbers involve shifting, masking, and arithmetic operations. Efficient implementations of such algorithms minimize computation of things like the masks and the number of bits to shift, which is pure overhead. In general, the results of such calculations change little during a simulation; the basic idea of program specialization is to perform such calculations at compile-time and hardwire them into the generated code. Especially for arithmetic algorithms, this can be very beneficial since the part of the algorithm that does actual work can be much smaller than the overhead computations.

3. Program Specialization

Program specialization, a class of partial evaluation [2, 7], starts with specialization predicates: aspects of the system that are known in advance, such as the number of bits in a fixed-precision number. A partial evaluator takes these specialization predicates and a program and performs a binding time analysis, which decides which parts of the program can be evaluated statically versus which must be evaluated while the system is running. Finally, the static portion of the program is evaluated and re-inserted in the generic program, producing a specialized program that observes only the dynamic inputs to the system.

I used static specialization—performed completely at compile-time—for these experiments. While there are also dynamic specialization techniques that generate code while the program is running, they are not as useful for simulating fixed-precision arithmetic since things such as word length do not tend to change.

Using program specialization, therefore, consists of identifying appropriate specialization predicates—a straightforward operation for fixed-precision arithmetic, coaxing the program specializer to generate code of the desired "shape," and integrating it back in the application.

Instructing a program specializer how to proceed can be tricky. To control Consel et al.'s [3, 4] Tempo program specializer, I used Prespec [14], a semi-graphical front-end tool developed for this purpose. Figure 4 shows a pair configuration files I gave to Prespec. Such files are typically written by the library designer to guide the program specializer. They indicate which variables should be treated as static (specializable) and the boundaries of the specialization process (function boundaries in Tempo). This example indicates the *multiply* function and the two functions it calls, *quantize* and *overflow*, should all be specialized. If *multiply*, say, called an error reporting function that we did not want to specialize because it is not performance-critical, we could indicate this by marking the error function as *extern* instead of *intern*.

The Prespec tool runs in two phases. Feeding the configuration file to the tool produces a specialization system. To this system, a user then provides details about the specialization scenario (e.g., specific wordlengths), runs the second half of Prespec, and this generates a C source file containing the specialized code. As a result, the user only needs to fill in a form describing how he or she would like to use the library components; the library designer has managed the details.

Consel et al. have applied program specialization to system software [15] such as TCP/IP stacks [1], which, like fixed-precision

```
Module fixed {  /* file fixed.mdl */
  Defines {
    From fixed.h {
      Fixed::struct fp {
        D(int) val;
        S(int) wl;
        S(int) iwl;
        S(int) lbp;
        S(int) overflow;
        S(int) rounding;
      };
    }
    From fixed.c {
      Quantize::intern
        quantize(Fixed(struct fp) S(*) r);
      Overflow::intern
        overflow(Fixed(struct fp) S(*) r);
    }
  }
  Exports { Fixed; Quantize; Overflow; }
}


Module mult { /* file mult.mdl */
  Imports {
    From fixed.mdl {
      Fixed;
      Quantize;
      Overflow;
    }
  }
  Defines {
     From fixed.c {
     Multiply::intern
       mult(Fixed(struct fp) S(*) r,
         Fixed(struct fp) S(*) a,
     Fixed(struct fp) S(*) b)
       { needs {
           Quantize;
           Overflow;
         }
       };
    }
  }
  Exports { Multiply; }
}
```

Figure 4. The Prespec [14] specializer configuration for the fixed-precision multiply routine in Figure 1. It defines all but the *val* field of the fixed-precision data type as taking static (specializable) values and says to assume the arguments to the multiply, quantize, and overflow functions also follow this pattern.

simulation libraries, are both performance-critical and written to handle many more cases than usually appear at run-time. They have the additional challenge of doing specialization while the system is running—dynamic specialization—since operating system code much cope with applications that are not known at compile-time. The experiments here do not use such a feature because the relevant specialization context is known before the system runs, although this could be used, say, for quickly comparing the behavior of the algorithm on many different word lengths.

Lawall [13] applies partial evaluation to speed the fast Fourier transform, one of the examples I describe below. Although her goal of speeding an implementation that uses processor-native floating-point differs substantially from mine (my focus is to speed arithmetic; hers is to speed its control skeleton), the 7.8× speed-up she obtained for the 16-point FFT is comparable to my result of 6.8× when also specializing the whole routine.

```
#include "systemc.h"
#define N 25

double fix_fir(double _in[])
{
  sc_fxtype_params param(32, 16, SC_RND, SC_SAT);
  sc_fxtype_context con(param);

  sc_fix in[N]; // Input samples
  sc_fix c[N];  // Coefficients
  sc_fix t[N];  // Temporary (intermediate) results
  sc_fix y;     // Accumulator

  // Initialize the coefficient vector
  int i;
  double ct = 0.9987966;
  for (i = 0 ; i<N ; i++) {
    in[i] = _in[i];
    c[i] = ct;
    ct /= 2;
  }

  // Compute the dot product
  for (i = 0 ; i < N ; i++) {
    t[i] = c[i] * in[i];
    y += t[i];
  }

  return y;
}
```

Figure 5. The original FIR routine, which uses the SystemC fixed-point libraries. Courtesy of Claire Fang.

4. Experimental Results

The main contribution of this paper is the following series of experiments. I applied program specialization to SystemC fixed-precision arithmetic libraries, which were not written with program specialization in mind, found that the results were about a magnitude worse than the FRIDGE results, then manually wrote a fixed-precision library designed for specialization that produces code like that from the FRIDGE system [10]. By design, this rewrite produced much faster code; specialization of the original libraries was only able to speed the SystemC code by a factor of two.

I experimented with three small but typical signal processing algorithms: FIR, a 25-point finite impulse-response filter that performs 25 multiply and add operations (Figure 5); IDCT, an 8-point inverse discrete cosine transform, comprising 18 additions, 12 subtractions, and 11 multiplies (Figure 7); and FFT, a 16-point fast Fourier transform (Figure 8). While real-world algorithms can be much larger than this, it is the size of the library, not the size of the algorithm, that is the main load on the program specializer.

Figure 5 shows the original C++ code for the FIR. Its simplicity is deceptive because all the usual arithmetic operators on the *sc_fix* type are overloaded and invoke costly routines. However, such simple top-level control-flow is typical of signal processing algorithms. To make this code palatable to Tempo, I manually rewrote it as shown in Figure 6. This consisted mostly of expanding out the function calls implicit in the overloaded operators, although I also split off the coefficient initialization code so I could run the actual FIR algorithm (the loop in the *fir* function that performs the multiply-and-accumulate operation) in isolation. The times I report do not include the time for initializing the *coeff* array, whose speed would not be critical in practice.

The code for the IDCT (Figure 7) is longer, but even more simple because it is basically a sequence of arithmetic operations. I do not show how I rewrote it in C because it is lengthy yet trivial.

```
#define INIT(a) \
  FIXED_INIT(a, 16, 4, 16, SATURATE, TRUNCATE)
#define N 25

void init_coeff(fixed *coeff)
{
  int i;
  fixed c, d2, tmp;

  INIT(c);
  FIXED_FROM_DOUBLE(c, 0.9987966);
  INIT(d2);
  FIXED_FROM_DOUBLE(d2, 0.5);
  INIT(tmp);

  for ( i = 0 ; i < N ; ++i ) {
    INIT(coeff[i]);
    FIXED_ASSIGN(coeff[i], c);
    FIXED_MUL(tmp, c, d2);
    FIXED_ASSIGN(c, tmp);
  }
}

void mac(fixed *acc, fixed *coeff, fixed *in)
{
  fixed sum, prod;
  sum.wl = 16;
  sum.iwl = 4;
  sum.lbp = 16;
  sum.overflow = SATURATE;
  sum.rounding = TRUNCATE;
  prod.wl = 16;
  prod.iwl = 4;
  prod.lbp = 16;
  prod.overflow = SATURATE;
  prod.rounding = TRUNCATE;
  FIXED_MUL(prod, *in, *coeff);
  FIXED_ADD(sum, *acc, prod);
  FIXED_ASSIGN(*acc, sum);
}

void
fir(fixed *result, fixed *coeff, fixed *in)
{
  int i;
  INIT(*result);

  for ( i = 0 ; i < N ; ++i) {
    mac(result, &coeff[i], &in[i]);
  }
}
```

Figure 6. My manually-recoded FIR routine.

The FFT routine, whose recoded form I show in Figure 8, is a bit more complicated, containing some nested loops but no conditionals. Again, this is typical of signal-processing code. Here I do not show the original SystemC source, which I took from the SystemC 2.0.1 distribution—it was one of the examples, because it was very long and messy; its core functionality and ultimate output is the same as for the code in Figure 8.

Table 1 lists the running times for the three examples implemented in eight different ways and the ratios of these times, i.e., the speed-up due to each coding improvement. The first six implementations listed in Table 1 produce identical results; the last two use the processor's native (hardware-accelerated) floating-point arithmetic instead of fixed-point and therefore produce slightly different results.

```c
#include "systemc.h"

double fix_idct(double _in[], double _out[]) {
  sc_fxtype_params param(32, 16, SC_RND, SC_SAT);
  sc_fxtype_context c(param);

  sc_fix x0, x1, x2, x3, y0, y1, y2, y3, z0, z1, z2, z3;
  sc_fix s0, s1, s2, s3, u0, u1, u2, u3, v0, v1, v2, v3;
  sc_fix t0, t1, t2, t3;
  sc_fix tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
  sc_fix in[8], out[8];

  for (int i = 0; i< 8; i++) in[i] = _in[i];

  sc_fix CONST1 = 1.414213562;
  sc_fix CONST2 = 1.306562965;
  sc_fix CONST3 = 0.541196100;
  sc_fix CONST4 = 0.125;
  sc_fix CONST5 = 0.899976223;
  sc_fix CONST6 = 2.562915448;
  sc_fix CONST7 = 0.601344887;
  sc_fix CONST8 = 0.509795579;

  x0 = in[1];          x1 = in[1] + in[3];
  x2 = in[3] + in[5];  x3 = in[5] + in[7];
  x1 = in[1] + in[3];

  y0 = x0 * CONST1;    y1 = x1 * CONST1;
  y2 = x2;             y3 = x1 + x3;

  z0 = y0 - y2;        z2 = y0 + y2;
  tmp0 = y1- y3;       z1 = tmp0 * CONST2;
  tmp1 = y1+ y3;       z3 = tmp1 * CONST3;

  tmp2 = z0 - z1;      tmp3 = z0 + z1;
  tmp4 = z2 - z3;      tmp5 = z2 + z3;
  s0 = tmp2 * CONST5;  s1 = tmp3 * CONST7;
  s2 = tmp4 * CONST6;  s3 = tmp5 * CONST8;

  u0 = in[0] - in[4];  u2 = in[0] + in[4];
  u1 = in[2] * CONST1; u3 = in[2] + in[6];

  v0 = u0;             v2 = u2;
  tmp6 = u1 - u3;      tmp7 = u1 + u3;
  v1 = tmp6 * CONST2;  v3 = tmp7 * CONST3;

  t0 = v0 - v1;        t1 = v0 + v1;
  t2 = v2 - v3;        t3 = v2 + v3;

  out[5] = (t0 - s0);  out[2] = (t0 + s0);
  out[6] = (t1 - s1);  out[1] = (t1 + s1);
  out[4] = (t2 - s2);  out[3] = (t2 + s2);
  out[7] = (t3 - s3);  out[0] = (t3 + s3);

  for (int i = 0; i< 8; i++)
    _out[i] = out[i];
}
```

Figure 7. The original IDCT routine, which uses the SystemC fixed-point libraries. Courtesy of Claire Fang.

```c
#define INIT(a) \
  FIXED_INIT(a, 16, 4, 16, SATURATE, TRUNCATE)

/*  N is points in FFT     2**M = N    */
#define M 4
#define N (1 << M)
#define LEN ((N)/2)

void fft(fixed supplied_sample_real[N],
         fixed supplied_sample_imag[N],
         fixed sample_real[N], fixed sample_imag[N],
         fixed W_real[LEN - 1], fixed W_imag[LEN - 1])
{
  int len, incr, windex, stage, i, j;
  fixed tmp1_real, tmp1_imag, tmp2_real, tmp2_imag;
  fixed prod1, prod2;

  INIT(tmp1_real);  INIT(tmp1_imag);
  INIT(tmp2_real);  INIT(tmp2_imag);
  INIT(prod1);      INIT(prod2);

  for ( i = 0 ; i < N ; ++i ) {
    FIXED_ASSIGN(sample_real[i],
                 supplied_sample_real[i]);
    FIXED_ASSIGN(sample_imag[i],
                 supplied_sample_imag[i]);
  }

  for ( len = N, incr = 1, stage = 0 ; stage < M ;
        incr *= 2, ++stage ) {
    len /= 2;

    // First iteration: no multiplication
    for ( i = 0 ; i < N ; i += len*2 ) {
      FIXED_ADD(tmp1_real, sample_real[i],
                sample_real[i+len]);
      FIXED_ADD(tmp1_imag, sample_imag[i],
                sample_imag[i+len]);
      FIXED_SUB(sample_real[i+len], sample_real[i],
                sample_real[i+len]);
      FIXED_SUB(sample_imag[i+len], sample_imag[i],
                sample_imag[i+len]);
      FIXED_ASSIGN(sample_real[i], tmp1_real);
      FIXED_ASSIGN(sample_imag[i], tmp1_imag);
    }

    // Remaining iterations: multiply by twiddle factors
    for ( windex = incr - 1, j = 1 ; j < len ;
          windex += incr, ++j ) {
      for ( i = j ; i < N ; i += len * 2 ) {
        FIXED_ADD(tmp1_real, sample_real[i],
                  sample_real[i+len]);
        FIXED_ADD(tmp1_imag, sample_imag[i],
                  sample_imag[i+len]);
        FIXED_SUB(tmp2_real, sample_real[i],
                  sample_real[i+len]);
        FIXED_SUB(tmp2_imag, sample_imag[i],
                  sample_imag[i+len]);
        FIXED_MUL(prod1, tmp2_real, W_real[windex]);
        FIXED_MUL(prod2, tmp2_imag, W_imag[windex]);
        FIXED_SUB(sample_real[i+len], prod1, prod2);
        FIXED_MUL(prod1, tmp2_real, W_imag[windex]);
        FIXED_MUL(prod2, tmp2_imag, W_real[windex]);
        FIXED_ADD(sample_imag[i+len], prod1, prod2);
        FIXED_ASSIGN(sample_real[i], tmp1_real);
        FIXED_ASSIGN(sample_imag[i], tmp1_imag);
      }
    }
  }
}
```

Figure 8. My recoded FFT routine, adapted from the SystemC 2.0.1 distribution.

Comparing the speed of fixed-point code with floating-point is somehow unfair, since the latter employs special-purpose hardware, but it does suggest a lower bound for execution time. Beating the speed of the floating-point implementation seems unlikely, but it is possible to come close. Compared with the best specialized code I was able to produce, the speedup for floating-point was only 2.8, 6.4, and 3.4 times for the three examples (eighth row in Table 1).

4.1 Specializing the SystemC Libraries

The fixed-precision arithmetic libraries in SystemC are vast, flexible, and fairly inefficient. Apparently designed to work on very large numbers (1000s of bits), they use a representation that avoids bit-level shifting at the expense of using more memory.

The main difficulty in specializing the SystemC libraries was the language in which they are written: C++. Tempo only accepts ANSI C, and I know of no C++ program specializer. I first tried to use a tool—Sun's gcc2c—to do this conversion automatically, but gcc2c's output violated too many good C coding practices and confused Tempo. For example, gcc2c represents every local variable as an integer and type-casts it when it is used.

Giving up on doing the C++-to-C translation automatically, I resorted to manually recoding the relevant routines. While I attempted to mimic the C++ implementation as closely as possible, this was not always possible. For example, a good optimizing C++ compiler can heed *const* declarations on methods to avoid calling functions multiple times (this is useful, for example, with the *is_zero()* method on fixed-precision numbers, which should return the same result provided the number is not modified). Similarly, the C++ libraries are written using many little methods that are inlined for efficiency; I manually inlined such code where possible.

The biggest change involved memory management. In an expression such as "$a = b + c$," two C++ operator functions are invoked: one that adds b and c to produce a new object, and one that copies that object to a. The SystemC libraries utilize an elaborate memory-management system that reuses objects. Rather than implement this, I instead wrote the "add" routine to take three arguments: two addends and a place to put the result. This probably accounts for most of the execution time difference between the SystemC libraries and the C reimplementation, which ranged from a factor of 3.4 to 6.6 (second row in Table 1).

While my re-coding was not automatic, I believe most of what I did for this phase could be automated. The C++ code used only the most basic object-oriented features (i.e., classes with no inheritance or virtual functions). Such code could be translated simply into C (such a program is mostly just a nice way of naming things plus additional type constraints), but writing such a tool was definitely beyond the scope of these experiments.

It turns out the main barrier in generating efficient specialized code from the SystemC libraries is their choice of number representation. They place the binary point of a number just to the right of a word (see Figure 9). As a result, the integer portion of a number always falls in a different word than the fractional, meaning most numbers are represented in two or more words. This is an interesting trade-off: while it means that bit-level shifts are never necessary to achieve alignment, it also means that operations on small numbers can require twice as many primitive operations. Specialization, unfortunately, is only able to recover some of this cost: since by definition it preserves the behavior of the original program, it cannot change the representation of the numbers and hence the number of primitive operations necessary.

Another inefficient aspect of the SystemC fixed-precision implementation is its use of an explicit sign bit. Maintaining this bit, which is stored as an integer with values ±1, is done explicitly and can cost as much as the multiplication of small mantissas. The cost


```
typedef unsigned long word;
typedef struct fixed {
  word *array;  /* mantissa array */
  int size;     /* words in the array */
  int q_mode;   /* Quantization mode */
  int o_mode;   /* Overflow mode */
  int state;    /* Current state of number */
  int wp;       /* units word index */
  int sign;     /* 1 or -1 */
  int msw;      /* most significant word */
  int lsw;      /* least significant word */
  int wl;       /* word length */
  int iwl;      /* integer word length */
} fixed_fix;
```
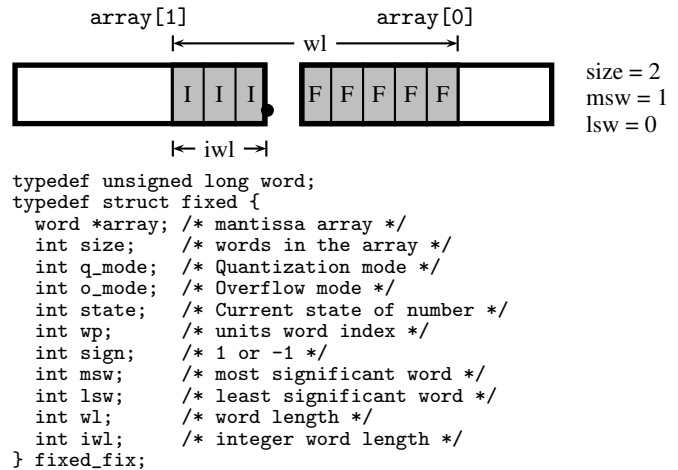
Figure 9. The representation of a fixed-precision number used in my C implementation of the SystemC libraries. Compared to Figure 3, bit-level shifting is never necessary in this representation because the binary point is always between words, but more words must be manipulated.

is higher for addition routines, which test sign bits and treat addends with different signs differently.

The biggest difficulty in efficient specialization of the SystemC libraries came from their use of "mostly static" variables. Specifically, the *lsw* and *msw* words, which index the least and most significant mantissa words, are constant for most numbers, but differ when representing zero, pure integers, and numbers less than one. When I recoded the algorithm, I made these values a function only of the word length, not the particular number being represented, so they could be specialized.

The SystemC libraries hit fundamental challenges in implementing multi-word fixed-precision arithmetic in C or C++. One problem is the absence of an integer multiplication operator that produces a double-word result. In C and C++, the product of two *int*s is itself an *int*; the extra bits that may be produced are simply discarded. As a result, the SystemC library resorts to multiplying half-words, quadrupling the number of multiplication operations that would otherwise be necessary. Surprisingly, the SystemC code does not take this into account and always performs multiplications as if the numbers consisted purely of whole words, a substantial waste of time for small fixed-precision numbers. I changed this when I rewrote the libraries in C, producing about a 20% speed-up for the (small) word size I was using.

The way the SystemC libraries used unions posed another challenge in adapting them for specialization with Tempo. The C++ code uses a union to pick apart a word into two half words. This is technically illegal in C definitely not supported by Tempo. I rewrote it to use explicit shifts.

The top three rows of Table 1 show the running times for the experiments using the original SystemC libraries, my recoding of them in (unspecialized) C, and the result of specializing these libraries with Tempo. All were compiled with *gcc -O* and run on a 2.53GHz Pentium 4 with a 512K cache and 512MB main memory running Linux.

The speed difference between the C++ SystemC libraries and my rewrite is substantial (the second row in Table 1: FIR: 4.2, IDCT: 6.6, FFT: 3.4). The very first C implementation, which omitted the memory-management system described above, provided a speed-up of about two. Unnecessary tests were removed in later versions, largely to improve the quality of the specialized code, but this also sped up the baseline.

Table 1. Execution times (in ns) and the corresponding speedups for the three examples.

	Times			Speedup vs. SystemC			vs. recoded in C			vs. Specialized			vs. for specialization			vs. library specialized			vs. program specialized			vs. double floats		
	FIR	IDCT	FFT																					
SystemC	26000	41000	110000	1	1	1																		
Recoded in C	6300	6100	34000	4.2	6.6	3.4	1	1	1															
Specialized	3700	3400	18000	7.1	12	6.1	1.7	1.8	1.8	1	1	1												
Written for specialization	2300	1500	8900	11	26	13	2.7	4	3.8	1.6	2.2	2.1	1	1	1									
Library specialized	1000	570	3400	25	72	33	6	11	9.8	3.6	5.9	5.4	2.2	2.7	2.6	1	1	1						
Program specialized	720	250	1300	36	160	86	8.7	24	26	5.2	13	14	3.2	6.1	6.8	1.4	2.2	2.6	1	1	1			
Double precision floats	290	40	420	92	1000	270	22	150	80	13	85	44	8	39	21	3.6	14	8.2	2.5	6.4	3.1	1	1	1
Single precision floats	260	40	380	100	1000	300	25	150	88	15	84	48	9	39	23	4.1	14	9	2.8	6.4	3.4	1.1	1	1.1

Only the "Times" numbers were measured; all other columns just report ratios of these numbers.

The speed-up gained by running single-precision floating-point compared to the specialized code varies more dramatically (the bottom row in Table 1: FIR: 15, IDCT: 84, FFT: 48). I attribute this to the number of non-fixed-precision operations in the examples and the general inefficiency of the arithmetic algorithms, which the specializer is not able to completely overcome. The IDCT example is straight-line arithmetic (no integer arithmetic or loops). FIR contains a loop around a small multiply-accumulate operation, and FFT has two nested loops.

The speed-up from specialization varies very little across examples (third row of Table 1: FIR: 1.7, IDCT: 1.8, FFT: 1.8) and comes from two factors: the improved speed of the arithmetic code and the overall mix of instructions. Since all three examples call the same specialized arithmetic routines, and the overhead in each example is small compared to the cost of the fixed-precision arithmetic, the speed-up is uniform across the examples.

4.2 Specializing a Custom Library

The initial goal of this work was to make the specialized techniques in FRIDGE more flexible without losing performance. Directly using the SystemC code could not achieve this because the specializer could not overcome the choice of number representation made by the SystemC library designers. So I wrote a library that uses the more-efficient representation in FRIDGE (Keding et al. [10]) and was able to achieve a much better speed-up, comparable to that of FRIDGE. Again, flexibility is the advantage of specialization over FRIDGE: algorithms can easily be changed without having to rewrite the code generator.

Keding et al. [10] represent fixed-precision numbers in a single machine word by placing the binary point to the left of the rightmost bit, as shown in Figure 3. Choosing such a representation reduces the number of words to represent small wordlength numbers and can reduce the number of primitive operations (e.g., +, *), but it does mean that add and multiply operations almost always require shifting before or after to ensure alignment (see the code in Figure 1). Fortunately, shifting is cheap on modern processors, so this representation provides a good trade-off for small word lengths.

The fourth, fifth, and six rows in Table 1 shows the execution times for these variants of the algorithms. I believe these are more representative of the power of specialization than the second and third because the code was written with specialization in mind.

The fifth and sixth rows in Table 1 show the effects of two types of specialization: for "Library specialized," the individual arithmetic functions were specialized in isolation, much as they were for the SystemC library. However, for "Program specialized," the entire algorithm, arithmetic functions included, was specialized. Obviously, the latter approach should produce faster code, if for no other reason than function inlining, at the expense of a larger executable. Specializing the arithmetic functions alone gives a better result (fifth row of Table 1: FIR: 2.2, IDCT: 2.7, FFT: 2.6) than

doing the same to the SystemC libraries since they were not written with specialization in mind.

The speed-up factor for specializing the algorithms along with the arithmetic functions (sixth row of Table 1: FIR: 3.2, IDCT: 6.1, FFT: 6.8) is substantially higher and represents the true power of program specialization. Unlike the case when only arithmetic functions are specialized, this also removes loops and virtually all integer arithmetic. The speedup is greatest for the FFT example, which contains a nested loop and substantial bookkeeping.

The varying speed gaps between specialized code and the single-precision floating-point implementation (row 8 in Table **??**: FIR: 2.8, IDCT: 6.4, FFT: 3.4) reflect the amount of non-fixed-precision arithmetic in the algorithms. IDCT (Figure 7) is pure arithmetic and thus closely represents the cost of floating-point vs. fixed-point arithmetic. The loops in the FIR and FFT floating-point implementation have not been specialized, so the difference is least for the FIR example, which has the most overhead. For similar examples, Keding et al. [10] report comparable numbers (factors of between 2.5 and 6.9).

5. Lessons for Library Writers

Writing a fast fixed-point library is not a simple task, but the additional burden of writing it for specialization seems reasonable. Such a library only needs to be written once for many applications to benefit from it, so it should be worth the additional effort.

In this section, I list guidelines for library authors who want to write code that can be effectively specialized for arithmetic-like algorithms. Writing other code such as interpreters for specialization demand more elaborate strategies [8, 9].

Understand the difference between static and dynamic code. This is the main lesson: a program specializer attempts to understand and specialize away every statement in a program, but it rarely can do so for every statement because a statement's behavior is often dependent on values that are only known at run time (e.g., the value of the samples passed to a signal-processing algorithm) or because the user has decided that it would be impractical to specialize a particular statement for every possible behavior it might have.

Maximize the fraction of static code. By definition, the static analyzer is able to account for the behavior of this code at compile-time and only its effects are carried over to the final generated code. Thus, the more the specializer can analyze, the less code there is in the final executable. There are two rules for doing this: try to increase the number of either truly constant variables, or the number of variables whose values can be tracked exactly by the specializer; and make sure the code itself is simple enough to be analyzed.

Write the static code for clarity and simplicity, not efficiency. Since static code is analyzed by the specializer and not executed when the program eventually runs, this code should be made

as easy to analyze as possible. This means, for instance, that it not rely on non-portable memory layout issues (e.g., using a *union* to split apart a word), complex pointer arithmetic, or other constructs that are risky and rely on complex emergent behavior. Instead, use arithmetic, arrays, and structures as much as possible.

Optimize the dynamic code. Much like how frequently-executed code in normal programs should be the focus of optimization, code that cannot be analyzed at compile-time by the specializer is performance-critical in the generated program because everything else is compiled away. Once you have established the distinction between static and dynamic code, make sure the dynamic code is as efficient as it can be. Clever algorithms and implementations of dynamic code is no more dangerous or unwise than in coding in a normal (non-specialized) environment.

Examine the code generated by the specializer. Unlike the output of an optimizing compiler, the output of a program specializer is usually quite readable because it is written in the high-level language. As a result, an iterative approach works well: write an initial algorithm, specialize it, examine the output, identify potential bottlenecks, and consider whether the operations could be moved into the specializer.

6. Conclusions

In this paper, I found program specialization could provide a 3–7× speed-up of fixed-precision arithmetic libraries. The speed of the generated code is comparable to that produced by the FRIDGE system [10], but the approach employed here does not require a dedicated fixed-precision arithmetic code generator. Instead, library designers write algorithms for generic components in their favorite language without having to write optimized code generation algorithms, yet still gain the speed advantages.

The C++ SystemC libraries did not specialize well, even after I performed a fairly mechanical translation into C to accommodate Tempo, a C-only specializer. The main problems were a poor choice of number representation that the specializer could not modify, and the use of "mostly static" variables, i.e., ones that very rarely changed, but that the specializer could not compile away.

A careful re-coding of the libraries, however, did specialize much better. The main change was the number representation, which I based on the representation described by Keding et al. [10, 11] designed for automatic code generation.

My conclusion is that reasonable performance from specialized code is only realistic when it was written with specialization in mind. This may be onerous for general programs, but seems reasonable for special-purpose code such as the fixed-point arithmetic libraries used in this experiment. The strategy for writing library code that specializes well is straightforward: the program should be thought of as a combination of static and dynamic variables and operators. Static variables and operators are "free" because they are evaluated completely at compile-time, but the dynamic aspects of the routine must be designed with efficiency in mind.

Debugging libraries written with specialization in mind is no more difficult than debugging ordinary libraries since the specializer guarantees that the semantics of the specialized code matches that of the original code. Optimizing the performance of such libraries is similarly easy: since the specializer produces fairly readable source code in the same language as the library, a library developer can iterate: generating specialized code, examining the generated code, and modifying the source as necessary to modifying the character of the generated code.

Experimental results show this methodology can speed carefully-written library code by a factor of between three and seven. While using such specialization as part of the development process is more work, most simulation users would find it a reasonable price to pay for nearly an order of magnitude in simulation speed.

References

[1] Sapan Bhatia, Charles Consel, Anne-Françoise Le Meur, and Carleton Pu. Automatic specialization of protocol stacks in OS kernels. In *Proceedings of the 29th Annual IEEE Conference on Local Computer Networks*, Tampa, Florida, November 2004.

[2] Charles Consel and Oliver Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 493–501, Charleston, South Carolina, January 1993.

[3] Charles Consel, L. Hornof, Julia L. Lawall, Renaud Marlet, G. Muller, J. Noyé, Scott Thibault, and E.-N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation (SOPE)*, 30(3es), September 1998.

[4] Charles Consel, Julia L. Lawall, and Anne-Françoise Le Meur. A tour of Tempo: A program specializer for the C language. *Science of Computer Programming*, 52(1–3):341–370, August 2004.

[5] Claire Fang Fang, Rob A. Rutenbar, Markus Püschel, and Tsuhan Chen. Toward efficient static analysis of finite-precision effects in DSP applications via affine arithmetic modeling. In *Proceedings of the 40th Design Automation Conference*, pages 496–501, Anaheim, California, June 2003.

[6] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer, Boston, Massachusetts, 2002.

[7] Neil D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–504, September 1996.

[8] Neil D. Jones. What *not* to do when writing an interpreter for specialisation. In *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 216–237, February 1996.

[9] Niel D. Jones. Transformation by interpreter specialisation. *Science of Computer Programming*, 52(1–3):307–339, August 2004.

[10] Holger Keding, Martin Coors, Olaf Lüthje, and Heinrich Meyr. Fast bit-true simulation. In *Proceedings of the 38th Design Automation Conference*, pages 708–713, Las Vegas, Nevada, June 2001.

[11] Holger Keding, Markus Willems, Martin Coors, and Heinrich Meyr. FRIDGE: a fixed-point design and simulation environment. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 429–435, Paris, France, March 1998.

[12] Donald E. Knuth. *The Art of Computer Programming*, volume two: Seminumerical Algorithms. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[13] Julia L. Lawall. Faster fourier transforms via automatic program specialization. In *Partial Evaluation—Practice and Theory, DIKU 1998 International Summer School*, volume 1706 of *Lecture Notes in Computer Science*, pages 338–355, Copenhagen, Denmark, June 1998.

[14] Anne-Françoise Le Meur, Julia L. Lawall, and Charles Consel. Specialization scenarios: A pragmatic approach to declaring program specialization. *Higher-Order and Symbolic Computation*, 17(1):47–92, March 2004.

[15] Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renauld Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems*, 19(2):217–251, 2001.