Using Program Specialization to Speed SystemC Fixed-Point Simulation

Stephen A. Edwards
Department of Computer Science, Columbia University
www.cs.columbia.edu/~sedwards
sedwards@cs.columbia.edu
Very simple, mathematically speaking:

\[ y_t = \sum_{i=0}^{n-1} a_i x_{t-i} \quad \text{n-tap FIR filter} \]

\[ S_n(u) = \frac{C_u}{2} \sum_{x=0}^{n-1} f(x) \cos \left( \frac{(2x + 1)\pi u}{16} \right) \quad \text{n-point IDCT} \]

\[ f_j = \sum_{k=0}^{n-1} x_k e^{-\frac{2\pi i j k}{n}} \quad \text{n-point DFT} \]
FIR in SystemC

```c
#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], c[N], t[N], y;

    int i; /* Init coefficients */
    double ct = 0.9987966;
    for (i = 0 ; i<N ; i++) {
        in[i] = _in[i]; c[i] = ct; ct /= 2;
    }

    for (i = 0 ; i < N ; i++) { /* Dot product */
        t[i] = c[i] * in[i]; /* Fixed-point multiplication */
        y += t[i]; /* Fixed-point addition */
    }

    return y; /* Type conversion */
}
```
What I Did

SystemC fixed-point code can be $1000 \times$ slower than floating-point. What can partial evaluation recover?

Selected Consel et al.’s Tempo + Prespec.

SystemC a C++ library; manually (stupidly) rewrote it in C.

Improvement from Tempo: $1.8 \times$.

Rewrote library for specialization.

Improvement from Tempo: $3–6 \times$

$3–6 \times$ slower than floating-point, comparable to Meyr et al.’s Fridge (a custom code generator)
Integers, Floating-point, Fixed-point

Integer

Floating-point

Fixed-point
SystemC’s Fixed-Point Types

`sc_fixed<8, 1, SC_RND, SC_SAT> fpn;`

8 is the total number of bits in the type
1 is the number of bits to the left of the decimal point
SC_RND defines rounding behavior
SC_SAT defines saturation behavior
Rounding Modes: SC_RND

Round up at 0.5
What you expect?
Rounding Modes: SC_RND_ZERO

Round toward zero
Less error accumulation
Rounding Modes: SC_TRN

Truncate
Easiest to understand
Overflow Modes: SC_SAT

Saturate
Sometimes desired
Overflow Modes: SC\_SAT\_ZERO

Set to zero
Odd Behavior
Overflow Modes: SC_WRAP

Wraparound
Easiest to implement
## Experimental Results

<table>
<thead>
<tr>
<th></th>
<th>Times</th>
<th>Speedup vs. SystemC</th>
<th>vs. recoded in C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FIR</td>
<td>IDCT</td>
<td>FFT</td>
</tr>
<tr>
<td>SystemC</td>
<td>26000</td>
<td>41000</td>
<td>110000</td>
</tr>
<tr>
<td>Recoded in C</td>
<td>6300</td>
<td>6100</td>
<td>34000</td>
</tr>
<tr>
<td>Specialized</td>
<td>3700</td>
<td>3400</td>
<td>18000</td>
</tr>
<tr>
<td>Doubles</td>
<td>290</td>
<td>40</td>
<td>420</td>
</tr>
<tr>
<td>Floats</td>
<td>260</td>
<td>40</td>
<td>380</td>
</tr>
</tbody>
</table>
The SystemC Representation

```c
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 */
    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;
```
The SystemC Representation

```c
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 */
    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;
```

Decimal point location avoids shifts but forces operations to manipulate two words.
The SystemC Representation

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 */
    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;

msw and lsw indicate which words are in use. Constant for almost all numbers, yet cannot be specialized.
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 */
    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;

Maintaining explicit sign bit as costly as mantissa for small numbers.
Other Challenges

In C, `int * int = int` (higher-order bits truncated)

32×32-bit multiplication: four mults plus masks & shifts.

SystemC libraries use a trick to convert to/from `double`:

```c
union ieee_double {
    double d;
    struct {
        unsigned negative : 1;
        unsigned exponent : 11;
        unsigned mantissa0 : 20;
        unsigned mantissa1 : 32;
    } s;
};
```

(Write to d, read from fields of s & vice versa)
Conclusion: Tempo cannot change representations, so can’t do much with SystemC code.

Solution: Try rewriting it for specialization.
typedef struct fp {
    int val;    /* 32-bit value */
    int wl;     /* Word length (bits) */
    int iwl;    /* Integer word length (bits) */
    int lbp;    /* Location of binary point (bits) */
    int overflow;
    int rounding;
} fixed;
typedef struct fp {
    int val;     /* 32-bit value */
    int wl;      /* Word length (bits) */
    int iwl;     /* Integer word length (bits) */
    int lbp;     /* Location of binary point (bits) */
    int overflow;
    int rounding;
} fixed;

Decimal point within a word requires shifting but permits single-word operations.
typedef struct fp {
    int val;    /* 32-bit value */
    int wl;     /* Word length (bits) */
    int iwl;    /* Integer word length (bits) */
    int lbp;    /* Location of binary point (bits) */
    int overflow;
    int rounding;
} fixed;

Only val is dynamic; everything else can be specialized.
The FRIDGE Representation

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

Two’s complement representation avoids additional sign field.
void mult(fixed *r, fixed *a, fixed *b) {
    int av, bv, shift;
    av = a->val >> (a->lbp - (a->wl - a->iwl));
    bv = b->val >> (b->lbp - (b->wl - b->iwl));
    shift = (a->wl - a->iwl) +
             (b->wl - b->iwl) - r->lbp;
    r->val = av * bv;
    if (shift > 0) r->val >>= shift;
    else if (shift < 0) r->val <<= -shift;
    fix_quantize(r);
    fix_overflow(r);
}
void quantize(fixed *r) {
    int shift, delta, mask;
    switch (r->rounding) {
        case ROUND:
            delta = 1 << (r->lbp - (r->wl - r->iwl)) - 1;
            shift = r->lbp - (r->wl - r->iwl);
            r->val = (r->val + delta) >> shift) << shift;
            break;
        case TRUNCATE:
            mask = 1 << (r->lbp - (r->wl - r->iwl)) - 1;
            r->val &= ~mask;
            break;
    }
}
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 &= 0xffffffff0; /* From quantize() */
    if (r->val > 0x7fff0) /* From overflow() */
        r->val = 0x7fff0;
    else if (r->val < -0x80011)
        r->val = -0x80011;
}

(wl=16, iwl=4, lbp=16, quant=TRUNC, overflow=SAT)
## Experimental Results

<table>
<thead>
<tr>
<th></th>
<th>Times</th>
<th>Speedup</th>
<th>vs. for specialization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FIR</td>
<td>IDCT</td>
<td>FFT</td>
</tr>
<tr>
<td>SystemC</td>
<td>26000</td>
<td>41000</td>
<td>110000</td>
</tr>
<tr>
<td>Rewritten</td>
<td>2300</td>
<td>1500</td>
<td>8900</td>
</tr>
<tr>
<td>Library specialized</td>
<td>1000</td>
<td>570</td>
<td>3400</td>
</tr>
<tr>
<td>Program specialized</td>
<td>720</td>
<td>250</td>
<td>1300</td>
</tr>
<tr>
<td>Double precision fbats</td>
<td>290</td>
<td>40</td>
<td>420</td>
</tr>
<tr>
<td>Single precision fbats</td>
<td>260</td>
<td>40</td>
<td>380</td>
</tr>
</tbody>
</table>
Conclusions

- Specializing “mechanical” C translation gave only a $1.8 \times$ speedup, still $22–150 \times$ slower than doubles
- Problem was poor choice of number representation
- Rewriting for specialization: $2.2–2.7 \times$ speedup
- Specializing program with libraries: $3.2–6.8 \times$
- Final result within a factor of $2.8–6.4 \times$ of doubles
- Not ready for prime time