# Fast Bit-True Simulation

Holger Keding, Martin Coors, Olaf Lüthje, Heinrich Meyr
Integrated Signal Processing Systems
Aachen University of Technology
Aachen, Germany

{keding,coors,luethje,meyr}@iss.rwth-aachen.de

## ABSTRACT

This paper presents a design environment which enables fast simulation of fixed-point signal processing algorithms. In contrast to existing approaches which use C/C++ libraries for the emulation of generic fixed-point data types, this novel approach additionally permits a code transformation to integral data types for fast simulation of the bit-true behavior. A speedup by a factor of 20 to 400 can be achieved compared to library based simulation.

## 1. INTRODUCTION

Algorithm design for digital signal processing systems typically starts in the floating-point domain to abstract from all implementation effects. On the other hand most implementations use fixed-point arithmetic due to the distinct advantage of fixed-point systems in terms of power consumption, chip size, and price per device.

Prior to the actual system implementation, a transformation from the floating-point to a fixed-point system is necessary, i.e an exploration of the fixed-point design space. Due to the non-linear nature of the quantization process, an exploration of the fixed-point design space with respect to quantization noise, performance, and operand word lengths can not be done without extensive system simulation. Thus slow fixed-point simulation would be a major bottleneck in the design flow.

The simulation of fixed-point systems is frequently done on a PC or a workstation utilizing a C/C++-based system-level design environment. For efficient modeling of finite word length effects language extensions implementing generic fixed-point data types are necessary. ANSI-C does not offer such data types and hence fixed-point modeling using pure ANSI-C becomes a very tedious and error prone task.

The language extensions implemented as libraries in C++ [1, 2, 3] offer a high modeling efficiency. They supply generic fixed-point data types and various casting modes for overflow and quantization handling. The simulation speed of these libraries on the other hand is rather poor.

Most C-based fixed-point libraries like the ETSI basic arithmetic operations [4] offer a set of two or three fixed-point data types and some data path[1] elements like they are frequently encountered on programmable DSPs. While this lack of flexibility restricts the applicability to a limited number of implementation platforms, the simulation speed is acceptable compared with the C++ based libraries. Still, there is a considerable overhead compared to an equivalent floating-point implementation.

Existing C++-based simulation libraries model the fixed-point operands as objects. In order to offer generic fixed-point data types without word length restrictions, data container types are used as an internal representation. Bit-true operations are performed by operator overloading. Range checking, the choice of cast modes and many other decisions necessary for correct bit-true behavior are done at simulation time. The price for this flexibility and ease of modeling is slow execution speed as the generic fixed-point data types modeled by extensive C++ constructs cannot be efficiently mapped to the architecture of the host machine by today's C++ compilers.

A simulation speedup can be achieved by mapping the fixed-point operands to the mantissa of the floating-point hardware of the host machine and bit level manipulations to maintain bit-true behavior. This restricts the maximum word length of the fixed-point operands to the word length of the mantissa. This approach has been described in by Kim [5] and it is also implemented in the SystemC library [3].

Another means of speeding up fixed-point simulations is the use of a hardware accelerator, e.g. an FPGA to perform computationally expensive operations. The acceleration can either be achieved by utilizing configurable logic or by combining configurable logic with a processor. This approach has been described by DeCoster [6]. The mapping of the algorithm to the different hardware units and the data transfer between the units make additional transformation steps necessary

The work described in this paper proposes a mapping of fixed-point algorithm in SystemC to an integer based ANSI C algorithm that directly addresses the built-in integer ALU of the host machine. An efficient mapping includes an embedding of all fixed-point operands into the host machine registers, a cast mode optimization and many other aspects, and require a detailed control and data flow analysis of the algorithm. Independently from the authors' work DeCoster [6] proposed a similar method, using DFL [7] as input

[1] A data path denotes the arithmetic units of a device, e.g. adders, multipliers, shifters.

language and targeting directly a Motorola DSP65000.

Our work presented here represents a continuation of the research results published by Keding et al. [8] and Willems [9] and introduces improved concepts for the mapping process, that result in a considerable simulation acceleration.

The paper is organized as follows: section 2 will briefly introduce the FRIDGE fixed-point design environment which forms the basis for the fast simulation techniques. In the main section 3 the transformation process is presented. In section 4 comparative benchmarking data for various signal processing kernel functions is provided. Section 5 concludes the paper.

## 2. FRIDGE DESIGN ENVIRONMENT

The work presented in this paper is based on the **F**ixed-Point P**R**ogrammIng and **D**esign **E**nvironment (FRIDGE) which supports the designer in the floating-point to fixed-point transformation process. FRIDGE is based on data flow analysis and information propagation which has been described in previous publications[10, 8]. The goal is to transform a signal processing algorithm into an entirely bit-true representation. The transformation is based on analytical range propagation of fixed-point operands and on simulation results. Fig. 1 highlights the design flow. Starting point is a floating-point

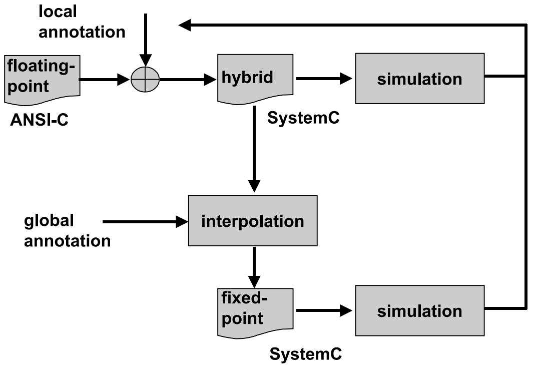

**Figure 1: Fixed-Point system design flow with FRIDGE**

description of the algorithm in ANSI-C. First the designer adds so called *local annotations* to the algorithm, specifying the fixed-point format for inputs and key operands. The local annotations are specified using the fixed-point data types of *SystemC*. The result is a hybrid description, i.e. parts are specified in fixed-point whereas the majority of the operands still remains floating-point. In a first step the FRIDGE front end parses in the hybrid description into a C++-based intermediate representation (IR). Then range propagation is performed to determine the bit-true format for all the operands. During this process, control- and data flow analysis is also carried out. The information gained is stored in the IR. The advanced algorithms used for the analysis have been described by Lüthje [11].

After this process the IR holds a bit-true description of the algorithm with additional control- and data flow information. These data structures form the basis for additional transformation steps performed in the FRIDGE back ends that target different languages and platforms. The *SystemC back end* transfers the IR into a fully quantized algorithm using the *SystemC* fixed-point data types This is beneficial for algorithm exploration since the desiger can easily reiew the results of the quantization. The methodology described here is also incorporated in the CoCentric Fixed-Point Designer tool [12].

For fast bit-true simulation, additional transformation steps are necessary. For the fast simulation back end we assume that fixed-point attributes are assigned to every operation. The back end also requires the information collected during the control- and data flow analysis stored in the IR. After a number of IR refinements, an ANSI-C representation of the algorithm using only integral data types can be derived from the IR. It is important to note that the transformation in the back end, in contrast to the float-to-fixed transformation in the IR, does not change the behavior of the algorithm. The fully quantized algorithm coded in *SystemC* and the integer-only ANSI-C algorithm yield bit-by-bit identical results, making the fast simulation back end output ideally suited for fast bit-true simulation on a workstation or PC.

## 3. TRANSFORMATION TO ANSI C

### 3.1 LBP Alignment

A fixed-point operand is specified by a triple *(wl,iwl,sign)* where *wl* is the *word length* of the operand, *iwl* the *integer word length* and *sign* the *sign encoding* used. For the embedding of this fixed-point operand into a register of the host machine with the machine word length *mwl* the minimum requirement is

$$mwl \geq wl = iwl + fwl \qquad (1)$$

Fig. 2 illustrates different options for embedding an operand with a word length of 5 bit into a given machine word length *mwl* of 8. Obviously for $mwl > wl$ a degree of freedom for choosing the **location of binary point** *lbp* exists:

$$mwl - iwl \geq lbp \geq wl - iwl = fwl \qquad (2)$$

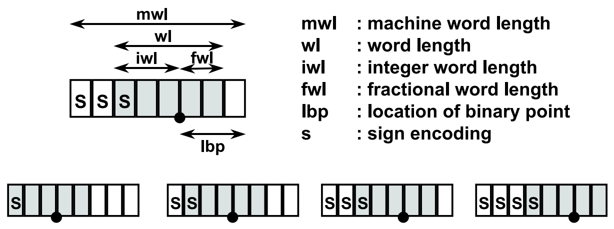
mwl
wl
iwl
fwl
lbp
s s s
mwl : machine word length
wl : word length
iwl : integer word length
fwl : fractional word length
lbp : location of binary point
s : sign encoding
s
s s
s s s
s s s s

**Figure 2: Embedding a 5-bit word into an 8-bit register**

Besides this degree of freedom there are also a number of limitations for the selection of the *lbp*:

- **Interface constraints**: for interface elements, e.g. function parameters or global variables the *lbp* must be defined identically for a function and all calls to this function. Otherwise the data written to or read from these data elements will be misinterpreted.

- **Operation constraints**: each operation has an *lbp syntax*. This *lbp syntax* may include constraints on the *lbp* of the operand(s) of the operation and/or rules for the calculation of the *lbp* of the result. E.g. the operands and the result of and addition must have the same *lbp*.

- **Control and data flow constraints**: generally a read access to a storage element must use the same *lbp* as the preceding write access to the storage element. This implies that if a write operation to a memory location occurs in alternative control-flow branches, the *lbp* must be at the same position in both write operations, as no run time information about the *lbp* is available in a following read operation. The same applies to ambiguous write operations to arrays and write operations via pointers.

### 3.1.1 The LBP Alignment Algorithm

The *lbp* alignment algorithm implemented in the fast simulation back end is designed to take advantage of the degree of freedom described by equation 2, while meeting the constraints specified above. For meeting these constraints and maintaining the consistency of the *lbps* one requires precise information about the control and data flow of the algorithm. To obtain this information we used the data flow analysis method described by Lüthje [11]. The data flow information is represented basically as *define-use (du) chains* and *use-define (ud) chains* [13, 14], but with additional and more accurate information about ambiguous control flow.

Initially for all operands *lbp=fwl* is chosen. Thus all operands are *right aligned.* In a first step we set the *lbps* of all interface elements according to the *interface constraints.*

Then, in an iterative process, the data flow information is used to adjust the *lbps* by insertion of shift operations to meet the *operation constraints* and the *control- and data flow constraints.* The algorithm terminates when all conditions are fulfilled and the *lbps* did not change during the last iteration.

The operation constraint *lbp* alignment algorithm basically consists of an iteration over all operations and an adjustment of the operand and result *lbps* according to the operation's *lbp syntax.*

The control- and data flow constraint *lbp* alignment algorithm searches for all read accesses from a data element the associated previous write accesses to the same data element. Using ud-chains this boils down to finding all *defines* for a *use* of a data element. According to the control and data flow constraints the *lbp* of operands linked by such ud-chains are set to the same value.

Finally, the embedding of *constants* can be done in a way that the required shift operations when using the constant are minimized.

Unlike described by Kum et al [15] we do not use a shift operation minimizing approach here. But using the degree of freedom in choosing a suited *lbp* (eq. 2) and the accurate data flow information we found that there is not sufficient potential for this optimization to justify the effort. An upper bound for a simulation speedup using shift minimization is given in section 4.

## 3.2 Data type Selection

The next step in the transformation process is the selection of suitable integral data types for fixed-point variables. The internal bit-true specification of the algorithm features arbitrary word lengths. With the SystemC back end this is no problem, since the SystemC data types are generic and may be of any bit length that is required. With the fast-simulation back end on the other hand one only has the limited pool of the built-in data types of the host machine, i.e. integral data types like `char`, `short`, `int`, `long`.

### 3.2.1 Basic constraints for any data element

A matching data type for every fixed-point variable has to be chosen. The minimum requirement for the data type chosen is that it can be embedded into the host machine data type with word length *mwl* at the correct location (see Fig. 2 for illustration):

$$iwl + lbp \leq mwl \tag{3}$$

### 3.2.2 Structural constraints

Additionally the requirements introduced by data structures that force each of their elements to be of the same data type have to be met An example for this behavior are arrays. The target data type for the $N$ elements of an array must fulfill the following condition:

$$MAX_{i=0}^{N-1}(iwl_{array}[i] + lbp_{array}[i]) \leq mwl \tag{4}$$

### 3.2.3 Semantical constraints

Another constraint becomes important if aliasing of data elements, e.g. by pointers occurs: a pointer may point to different data elements. For syntax and semantics reasons all aliased data elements and the base type of the pointer must be identical [16]. This only causes a problem if data types are changed like it is done in fixed-point optimizations or the floating-point to fixed-point transformation process described in section 2: initially most numerical data types are floating-point types but after the transformation there are various different fixed-point data formats. Hence special care must be taken during the code generation process that the types are consistent.

*Definition 1.* All data elements that may be aliased by a pointer and all pointers that may alias these data elements form an *alias DAG* [2]. The data elements form the leaves of the DAG while the pointers form the non-leaf nodes of the DAG. If there is one pointer that can be an alias for every data element this pointer represents the root node of an *alias TREE.*

The following code is an example for such an alias DAG:

```
float *p,*q,a[3],b[7],c[10];

if (condition) {p = a; q = b}
else           {p = b; q = c}

*p = *q = 1.75;
```

The pointer `p` is an alias for the elements of the arrays `a` and `b` while the pointer `q` is an alias for the arrays `b` and `c`. The resulting DAG is depicted in Fig. 3.

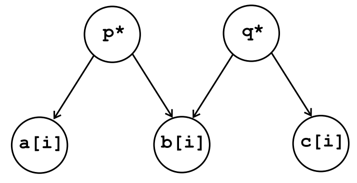

**Figure 3: Alias DAG: p and q are aliases for the arrays a, b, and c**

The correct data type for the elements of an alias DAG is then selected by a recursive maximum-data-type search over

[2]A DAG is a directed acyclic graph

the nodes of the DAG: First we pick an arbitrary node and we supply the size of the smallest data type available on the host machine to this node. The node marks itself as *visited* and determines the maximum of the supplied data type size and its own data type size. Then it supplies the result to all neighboring nodes that are not visited yet. This way the entire DAG is traversed and one receives the correct data type.

Alternatively one could imagine an algorithm for *pointer splitting* or *de-aliasing*, i.e. to modify the code and to introduce additional pointers so one ends up with pointers that do not point to multiple data elements any longer. Nevertheless in the field of DSP oriented algorithms we have not seen any significant improvements with this technique.

#### 3.2.4 Multiple-Precision Arithmetic

SystemC features data types of arbitrary bit width - which is rather uncommon for the registers of a processor. Most general purpose processors contain 32-bit or 64-bit integer units. We denote the largest C-addressable word as *fullword* with a bit width of length $mwl_{max}$. So, of course, the question arises what happens if we need to transform a SystemC algorithm containing operands with $lbp+iwl > mwl_{max}$ into ANSI C.

There are a couple of *multiple precision arithmetic libraries* available that represent a viable solution to this problem - one of the best best suited we encountered is the *GNU multiple precision arithmetic library (GMP)* [17]. On the other hand even the fastest of those libraries are also designed for ease of use and hence maintain generic types and functions, which also comes with a certain run-time penalty.

Our primary goal on the other hand is to optimize for speed, i.e. to skip any generic element in the generated code if it slows down the simulation. Hence we analyze at code-generation-time what precision and which fixed-point operations are needed and produce the code only containing the very necessary elements. For multiple precision we use a container based *multiple-precision arithmetic*, where one operand is stored in two or more fullwords and with accordingly modified operations. A good description of the principles of multi precision arithmetic is provided by Knuth [18].

## 3.3 Cast Mode Transformation

Cast operations can reduce or limit the wordlength on the MSB side of a word (*overflow handling*) or at the LSB side of a word (*quantization handling*). They are used either to prevent indeterministic behavior of fixed-point systems [3] or to model a data path that is different from the host machine. This is often the case when algorithms for DSP systems are developed. Fixed-point libraries like in SystemC offer various generic overflow and quantization handling modes which make them an efficient means of modeling fixed-point systems. For fast fixed-point simulation on the other hand the use of these generic casting modes are simply ruled out for performance reasons.

#### 3.3.1 Overflow Handling

Overflow handling is required if it is necessary to reduce the $wl$ at the MSB side of the word or if the carry bit is set for the MSB. Examples for frequently used overflow handling modes in digital signal processing algorithms are *wrap-around* and *saturation* [19].

**Saturation:** In SystemC a cast of an expression *expr* to a $wl$-bit two's complement ($tc$) data type with integer word length $iwl$ applying saturation as overflow mode can be modeled as follows:

```
result = sc_fix(expr,wl,iwl,...,SC_SAT);
```

The fast simulation code generation on the other hand translates this into plain C code that first tests if the range of data type is exceeded, and if so it sets the resulting value to the minimum or maximum of this type, which is:

$$\begin{aligned} MAX_{wl,iwl,lbp,tc} &= 2^{iwl+lbp-1} - 2^{lbl-fwl} && (5)\\ MIN_{wl,iwl,lbp,tc} &= -2^{iwl+lbp-1} + 2^{lbl-fwl} - 1 && (6)\\ with\ wl &= iwl + fwl && (7)\end{aligned}$$

Thus the fast simulation code construct generated is [4]:

```
int tmp;
result = ((tmp=expr)>MAX)?MAX:(tmp<MIN)?MIN:tmp;
```

Introducing an additional temporary variable avoids multiple evaluations of `expression`.

**Wrap-Around:** The SystemC way of casting an expression *expr* to a $wl$-bit two's complement ($tc$) data type with integer word length $iwl$ applying wrap-around as overflow mode is shown here:

```
result = sc_fix(expr,wl,iwl,...,SC_WRAP);
```

For the bit-true ANSI-C equivalent of this operation several options exist. An example for a code construct for *wrap around* assuming two's complement arithmetic and a machine wordlength of $mwl$ is:

```
result = (expr << SHIFT) >> SHIFT;
```

The amount of shifts computes to $SHIFT = mwl - iwl - lpb$. The shift left eliminates the MSBs whereas the arithmetic shift right provides a sign extension for the new MSB.

#### 3.3.2 Quantization Handling

If the word length of an operand is reduced at the LSB side one can apply different quantization handling modes. The most frequently encountered are *rounding* and *truncation*.

**Rounding:** In SystemC the method for casting an expression *expr* to a $wl$-bit two's complement data type with integer word length $iwl$ applying rounding as quantization mode is:

```
result = sc_fix(expr,wl,iwl,SC_RND,...);
```

Rounding is defined by adding $DELTA = LSB/2$ to the operand and eliminating the LSBs, e.g. by shifting it right $SHIFT$ bits. With

$$\begin{aligned} DELTA &= 2^{lbp-fwl-1} && (8)\\ SHIFT &= lbp - fwl && (9)\end{aligned}$$

Thus the rounding operation can be realized in the fast simulation code by:

```
result = ((expr + DELTA)>>SHIFT)<<SHIFT;
```

[3] In many cases the ANSI-C standard [16] does not specify the bit-true behavior of integral data types in case of overflow, quantization, etc.

[4] Note that for the code generation we also take the bit-true properties of the processor and compiler into account

**Truncation:** The truncation operation, given in SystemC by

```
result = sc_fix(expr,wl,iwl,SC_TRN,...);
```

can be implemented efficiently by a bit mask operation:

```
result = expr & (~MASK);
```

Where $MASK$ is given by $2^{lpb-fwl-1}$

For several combinations of cast modes, e.g. wrap-around combined with rounding or truncation, efficient joint fast simulation C-code constructs are generated. The shift operations introduced by the cast code constructs are also utilized to adjust the *lbp* of the expression, eliminating the need for additional scaling shifts.

## 4. EXPERIMENTAL RESULTS

The code generated by the FRIDGE fast simulation back end has been benchmarked against the fixed-point simulation classes which are part of the *SystemC* language. The *SystemC* implementation of the generic fixed-point data types is based on C++. The simulation classes offer two simulation modes: a mode supporting unlimited fixed-point word lengths based on concatenated data containers and a mode supporting limited precision up to 53 bits based on float-arithmetic and bit manipulations.

The benchmarks have been performed on a SUN Ultra 10 workstation running SOLARIS using the GCC compiler version 2.95.2 with the `-O3` option. The *SystemC* library version 1.0 was utilized for the bit-true simulations. The benchmark is based on typical signal processing kernels:

- **FIR** 17-tap FIR filter
- **DCT** 8x8 JPEG DCT algorithm
- **Autocorr** 25 elements 5th order autocorrelation
- **IIR** 3rd order IIR filter
- **FFT** complex FFT of length 8
- **Matrix** 4x4 matrix multiplication

Four different versions of the kernel functions have been benchmarked:

- **Floating-Point** The execution speed of the floating-point implementation of the algorithms serve as reference for the benchmarks

- **SystemC** The quantized bit-true version of the algorithms utilizing the *SystemC* fixed-point data types. The algorithms have been quantized using the *FRIDGE* design environment.

- **SystemC limited precision** The quantized bit-true code has been compiled with the *limited precision option* to speed up *SystemC* fixed-point operations.

- **Fast Simulation Code** The fast fixed-point simulation code based on integral data types has been generated by the FRIDGE back end applying the transformation techniques described in the previous sections. The code yields bit-by-bit the same results as the code utilizing the *SystemC* data types.

The experimental results are presented in Table 1. As the floating-point code has been used as a reference, the experimental data has been scaled relative to the execution speed of the floating-point code. The bit-true *SystemC* code consumes by a factor of 325...1103 more runtime than the original floating-point code, making bit-true simulation a major bottleneck in the fixed-point design flow. By utilizing the *limited precision* mode of the *SystemC* library, a speedup by a factor of 3.1...5.2 can be achieved, but the fixed-point code is still by a factor of 67...234 slower than the floating-point reference.

The speedup gained by running the fast simulation code generated by the FRIDGE fast simulation back end compared to the *SystemC* code is illustrated in Figure 4. The fast simulation code runs by a factor of 18.8...90.9 faster compared to the *SystemC* fixed-point code utilizing the *limited precision* option. For the *unlimited precision* the speedup is 91.0...454.2 respectively.

Compared to the floating-point reference code, the fast simulation code is by a factor of 2.5...6.9 slower. This is due to the host system's architecture and additional shift and bit mask operations necessary to perform *lbp*-alignment and cast operations to maintain bit-by-bit consistency with the quantized code.

The quantized *DCT* algorithm contains many *sc_fix* operations to reduce fixed-point word lengths introduced by the quantization process. As these operations can be modeled efficiently by bit mask operations in the fast simulation code, the highest speedup was achieved for this kernel function.

| | Floating-Point ANSI-C | SystemC | SystemC Limited Precision | Fast Simulation Code |
|---|---|---|---|---|
| **FIR** | 1.0 | 386.5 | 102.7 | 2.8 |
| **DCT** | 1.0 | 1103.1 | 233.9 | 2.5 |
| **Autocorr** | 1.0 | 694.6 | 130.6 | 6.9 |
| **IIR** | 1.0 | 371.0 | 120.2 | 3.1 |
| **FFT** | 1.0 | 354.7 | 67.7 | 2.6 |
| **Matrix** | 1.0 | 325.9 | 71.2 | 3.6 |

**Table 1: Relative Execution Speed**

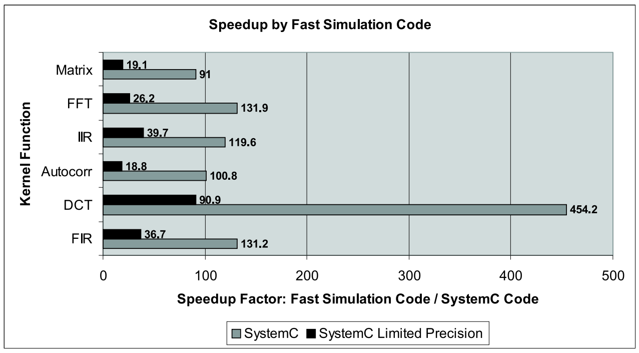

**Figure 4: Speedup by Fast Simulation Code**

For an estimate of the upper bound of speedup achievable by shift and cast optimization, we have benchmarked different derivate of the fast simulation code:

As laid out Willems [9], it is possible to model the *lbp* alignment by an ILP (**I**nteger **L**inear **P**rogramming) problem. A commercial ILP solver package can be used to find optimal solutions for the embedding of the operands utilizing a minimum number of shift operations to align *lbps* . Unfortunately solving ILP problems requires exponential effort with growing problem size which yields very long run times for the solver package. An upper bound on the efficiency of shift optimization is given by a solution requiring no scaling shifts at all. In order to get quantitative data for this upper bound we have removed all scaling shifts from the fast simulation code for the benchmarking kernels. (Which then of course yields incorrect numerical results)

Similar considerations apply for the number of cast operations in the fixed-point code. An upper bound for the efficiency of cast minimization is given by the performance of a fixed-point code which contains no cast operations at all. Thus we have benchmarked fast simulation code for the kernels which had all scaling shifts and all cast operations removed.

The experimental results are presented in Table 2. It is noteworthy that on the SUN Ultra 10 workstation utilized for the benchmarks the integer-only fast simulation code containing no scaling shifts and no casts runs by a factor of 1.39...2.76 slower than the original floating-point reference code. For the algorithms benchmarked, the potential of shift optimization is limited to 3%...13% . Similarly the potential of cast minimization is limited to 15%...35%

| | **Floating-Point ANSI-C** | **Fast Simulation Code** | **Fast Simulation Code no shift** | **Fast Simulation Code no shift no cast** |
|---|---|---|---|---|
| **FIR** | 1.0 | 2.79 | 2.71 | 2.45 |
| **DCT** | 1.0 | 2.46 | 2.15 | 1.39 |
| **IIR** | 1.0 | 3.13 | 3.09 | 2.76 |
| **FFT** | 1.0 | 2.61 | 2.58 | 2.21 |

**Table 2: Integer Code Performance**

## 5. CONCLUSIONS

In this paper we have presented a novel approach to accelerate bit-true simulation by directly mapping fixed-point operands to the built-in integral data types of the host machine. The concept and the necessary code transformation steps have been presented. As a proof of concept we have implemented the algorithms described in a software tool and we have benchmarked the generated ANSI-C code against the floating-point reference code and the *SystemC* fixed-point simulation classes. A speedup by a factor of 20 to 400 compared to the *SystemC* code while maintaining bit-by-bit equivalence was achieved. The fast simulation code generation benefits directly from the advanced control- and data flow analysis performed in the FRIDGE design environment during the interpolative transformation.

Future research work will focus on the generation of optimized C-code for dedicated fixed-point DSPs exploiting the architecture and the features of the target hardware and of the DSP C-compiler.

## 6. REFERENCES

[1] S. Kim, K. Kum, and W. Sung, "Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs," in *Workshop on VLSI and Signal Processing '95*, (Osaka), pp. 197–206, Nov. 1995.

[2] Frontier Design Inc., 9000 Crow Canyon Rd., Danville, CA 94506, USA, *A|RT Library User's and Reference Documentation*, 1998.

[3] Synopsys, Inc., CoWare, Inc., Frontier Design Inc., *SystemC User's Guide, Version 1.0*, 2000.

[4] Recommendation GSM 06.10, *GSM Full Rate Speech Transcoding*. ETSI/TC SMG, 1992.

[5] S. Kim, K. Kum, and W. Sung, "Fixed-Point Optimization Utility for C and C++ Based Digital Signal Process ing Programs," *IEEE Transactions on Circuits and Systems II*, Nov. 1998.

[6] Luc De Coster, *Bit-True Simulation of Digital Signal Processing Applications*. PhD thesis, KU Leuven, 1999.

[7] Mentor Graphics, *DSP Architect, DFL User's and Reference Manual*, 1994.

[8] H. Keding, M. Willems, M. Coors, and H. Meyr, "FRIDGE: A Fixed-Point Design and Simulation Environment," in *Proceedings of the European Conference on Design, Automation and Test (DATE)*, (Paris), pp. 429–435, Feb. 1998.

[9] M. Willems, *A Methodology for the Efficient Design of Fixed-Point Systems*. PhD thesis, Aachen University of Technology, 1998. in German.

[10] M. Willems, V. Bürsgens, H. Keding, T. Grötker, and H. Meyr, "System Level Fixed-Point Design Based on an Interpolative Approach," in *Proceedings of the Design Automation Conference (DAC)*, (Anaheim), pp. 293–298, Jun. 1997.

[11] O. Lüthje, H. Keding, and M. Coors, "High Performance Code Analysis by Abstract Execution," in *DSP Germany 2000 Proceedings, Munich*, Oct. 2000. in German.

[12] Synopsys, Inc., 700 E. Middlefield Rd., Mountain View, CA 94043, USA, *CoCentric Fixed-Point Designer - User's Manual.*

[13] M. J. Wolfe, *High Performance Compilers for Parallel Computing*. Redwood City, CA: Addison-Wesley Publishing, 1996.

[14] A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[15] K. Kum, J. Kang, and W. Sung, "A Floating-Point to Integer C Converter with Shift Reduction for Fixed-Point Digital Signal Processors," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2163–2166, 1999.

[16] B. W. Kernighan and D. M. Ritchie, *The C Programming Language (second edition)*. Prentice Hall, 1988.

[17] Granlund, T., *The GNU Multiple Precision Arithmetic Library*. Free Software Foundation, Boston, MA, USA, 3.1 ed.

[18] D. Knuth, *The Art of Computer Programming, Seminumerical Algorithms*, vol. 2. Addison-Weseley, second ed., 1984.

[19] S. K. Mitra, *Digital Signal Processing: A Computer-Based Approach*. New-York: McGraw-Hill, 1998.