Hindawi Publishing Corporation
EURASIP Journal on Applied Signal Processing
Volume 2006, Article ID 96421, Pages 1–19
DOI 10.1155/ASP/2006/96421

Floating-to-Fixed-Point Conversion for Digital Signal Processors

Daniel Menard, Daniel Chillet, and Olivier Sentieys

R2D2 Team (IRISA), ENSSAT, University of Rennes I, 6 rue de Kerampont, 22300 Lannion, France

Received 1 October 2004; Revised 7 July 2005; Accepted 12 July 2005

Digital signal processing applications are specified with floating-point data types but they are usually implemented in embedded systems with fixed-point arithmetic to minimise cost and power consumption. Thus, methodologies which establish automatically the fixed-point specification are required to reduce the application time-to-market. In this paper, a new methodology for the floating-to-fixed point conversion is proposed for software implementations. The aim of our approach is to determine the fixed-point specification which minimises the code execution time for a given accuracy constraint. Compared to previous methodologies, our approach takes into account the DSP architecture to optimise the fixed-point formats and the floating-to-fixed-point conversion process is coupled with the code generation process. The fixed-point data types and the position of the scaling operations are optimised to reduce the code execution time. To evaluate the fixed-point computation accuracy, an analytical approach is used to reduce the optimisation time compared to the existing methods based on simulation. The methodology stages are described and several experiment results are presented to underline the efficiency of this approach.

1. INTRODUCTION

Most embedded systems integrate digital signal processing applications. These applications are usually designed with high-level description tools like CoCentric (Synopsys), Matlab/Simulink (Mathworks), or SPW (CoWare) to evaluate the application performances with floating-point simulations. Nevertheless, if digital signal processing algorithms are specified and designed with floating-point data types, they are finally implemented into fixed-point architectures to satisfy the cost and power consumption constraints associated with embedded systems. In fixed-point architectures, memory and bus widths are smaller, leading to a definitively lower cost and power consumption. Moreover, floating-point operators are more complex to process the exponent and the mantissa. Thus, floating-point operator area and latency are greater compared to fixed-point operators.

In this context, the application specification must be converted into fixed-point. The manual conversion process is a time-consuming and an error-prone task which increases the development time. Some experiments [1] have shown that this manual conversion can represent up to 30% of the global implementation time. To reduce the application time-to-market, high-level development and code generation tools are needed. Thus, methodologies for automatic floating-to-fixed-point conversion are required to accelerate the development.

For digital signal processors (DSPs), the methodology aim is to define the optimised fixed-point specification which minimises the code execution time and leads to a sufficient accuracy. For this accuracy, the desired application performances must be reached. Existing methodologies [2, 3] achieve a floating-to-fixed-point transformation leading to an ANSI-C code with integer data types. Nevertheless, the data types supported by the DSP and the processor scaling capabilities are not taken into account to determine the fixed-point specification. The analysis of the architecture influence on the computation accuracy underlines the necessity to take the DSP architecture into account to optimise the fixed-point specification [4]. Furthermore, the code generation and the conversion process must be coupled.

In this paper, a new methodology to implement floating-point algorithms in fixed-point processors under accuracy constraint is presented. Compared to the existing methods, the processor architecture is taken into account and the floating-to-fixed-point conversion process is coupled with the code generation process. The fixed-point specification is optimised to reduce the code execution time as long as the application performances are reached. These optimisations are achieved through the location of the scaling operations and the selection of the data word-length according to the different data types supported by recent DSPs. The scaling operations are moved to reduce the code execution time. This paper is organised as follows. The previous works for the floating-to-fixed-point conversion are presented in Section 2. Our methodology is detailed in Section 3. For the different methodology stages, our approach is justified and the technique used to solve the problem is described. Finally, in Section 4, different experiments are presented underlining the efficiency of our approach.

2. RELATED WORKS

2.1. Floating-to-fixed-point conversion methodologies

In this section the different available methodologies for the automatic implementation of floating-point algorithms into fixed-point architectures are presented.

In [5], a methodology which implements floating-point algorithms into the TMS320C25/50 fixed-point DSP (Texas Instruments) is proposed. The floating-to-fixed-point conversion is achieved after the code generation process. This methodology is specialised for this particular architecture and cannot be transposed to other architecture classes.

The two methodologies presented below achieve the floating-to-fixed-point transformation at the source code level. The FRIDGE [6] methodology, developed at the Aachen University, transforms the floating-point C source code into a C code with fixed-point data types. In the first step, called *annotations*, the user defines the fixed-point format of some variables which are critical in the system or for which the fixed-point specification is already known. Moreover, global annotations can be defined to specify some rules for the entire system (maximal data word-length, casting rules). The second step, called *interpolation* [6, 7], determines the application fixed-point specification. The fixed-point data formats are obtained from a set of propagation rules and the analysis of the program control flow. This description is simulated to verify if the accuracy constrains are fulfilled. The commercial tool *CoCentric Fixed-point Designer* proposed by Synopsys is based on this approach.

In [3] a method called *embedded approach* is proposed to generate an ANSI-C code for a DSP compiler from the fixed-point specification. The data (source data), for which the fixed-point formats have been obtained with the technique presented previously, are specified with the available data types (target data) supported by the target processor. The degrees of freedom due to the source data position in the target data are used to minimise the scaling operations. This methodology produces a bit-true implementation into a DSP of a fixed-point specification. But accuracy and execution time are not optimised through the fixed-point format modification of some relevant variables.

The aim of the tool presented in [2, 8] is to transform a floating-point C source code into an ANSI-C code with integer data types. This code is independent of the targeted architecture. Moreover, a fixed-point format optimisation is done to minimise the number of scaling operations. Firstly, the floating-point data types are replaced by fixed-point data types and the scaling operations are included in the code. The scaling operations and the fixed-point data formats are determined from the dynamic range information obtained with a statistical method [9]. The reduction of the scaling operations number is based on the assignation of a common format to several relevant data to minimise the scaling operations cost function. This cost function takes account of the number of each scaling operation occurrences and depends on the processor scaling capabilities. For a processor with a barrel shifter, the cost of a scaling operation is set to one cycle; otherwise the number of cycles required for a shift of n bits is equal to n cycles.

This methodology achieves the floating-to-fixed-point conversion with the minimisation of the scaling operations cost. But, the code execution time is not optimised under a global accuracy constraint. The accuracy constraint is only specified through the definition of a maximal acceptable accuracy degradation allowed for each data. The data types supported by the architecture are not taken into account to optimise the fixed-point data formats. Moreover, the architecture model used to minimise the scaling operations number is not realistic. Indeed, for conventional DSPs including a barrel shifter and based on a MAC (multiply-accumulate) structure, the scaling operation execution time depends on the data location in the data path and is not always equal to one cycle. Furthermore, for processors with instruction-level parallelism (ILP) capabilities, the overhead due to scaling operations depends on the scheduling step and cannot be easily evaluated before the code generation process.

Compared to these methods, our approach optimises the data word-length to benefit from the different data types supported by recent DSPs. Moreover, the scaling operation location is optimised with a realistic model to evaluate the scaling operation execution time. The goal of these two optimisations is to minimise the code execution time as long as the accuracy constraint is fulfilled. In our methodology, the processor architecture is taken into account and the floating-to-fixed-point conversion process is coupled with the code generation process.

2.2. Fixed-point accuracy evaluation

Despite fixed-point computation, the application quality criteria must be verified. Thus, the computation accuracy due to fixed-point arithmetic is evaluated. Most of the available methodologies are based on a bit-true simulation of the fixed-point application [10–12]. Nevertheless, this technique suffers from a major drawback which is the time required for the simulations [11]. The fixed-point mechanism emulation on a floating-point workstation increases the simulation time compared to a classical floating-point simulation. Moreover, a great number of samples is required to verify if the application quality criteria are respected. This drawback becomes a severe limitation when these methods are used in the process of fixed-point optimisation where multiple simulations are needed to explore the design space [10]. For each evaluation of the fixed-point specification accuracy, a new simulation is required.

An alternative to the simulation-based method is the analytical approach. The verification that the fixed-point implementation respects the application quality criteria is achieved in two steps with the help of a single metric. The most commonly used metric to evaluate the computation accuracy is the signal-to-quantisation-noise ratio (SQNR) [10, 13, 14]. This metric defines the ratio between the desired signal power and the quantisation noise power. Thus, first of all, the minimal value of the computation accuracy (SQNR_{min}) is determined and then, the fixed-point specification is optimised under this accuracy constraint. The accuracy constraint (SQNR_{min}) is determined according to the application performance constraints. The main advantage of the analytical approach is the execution time reduction of the fixed-point optimisation process. Indeed, the SQNR expression determination is done only once, then, the fixed-point system accuracy is evaluated through the computation of a mathematical expression.

In our methodology, an analytical approach is used to evaluate the computation accuracy. This approach [14] reduces significantly the execution time of the fixed-point optimisation process, compared to the simulation-based methods. This method is described with further details in Section 3.1.3.

3. FLOATING-TO-FIXED-POINT CONVERSION METHODOLOGY

The aim of the methodology presented in this paper is to implement automatically a floating-point application into a fixed-point DSP. Despite the computation error due to the fixed-point arithmetic, the different quality criteria (performances) associated with the application must be respected. For embedded systems, the cost and the power consumption must be minimised. Thus, the optimised fixed-point specification which minimises the code execution time and fulfils a given computation accuracy constraint must be determined. To optimise the implementation, the targeted architecture must be taken into account during the fixed-point conversion process.

3.1. Methodology flow

The methodology flow has been defined from the analysis of the architecture influence on the computation accuracy and from the study of the interaction between the fixed-point conversion process and the code generation process. The global methodology flow is presented in Figure 1. The tool is made up of two main blocks corresponding to the compilation infrastructure and to the floating-to-fixed-point conversion.

The compilation infrastructure front-end generates an intermediate representation from the floating-point C source code. The floating-to-fixed-point conversion process is applied on this intermediate representation. The assembly code is generated with the compilation infrastructure back-end from this fixed-point intermediate representation.

The first stage of the fixed-point conversion process corresponds to the data dynamic range evaluation. These results are used to determine the data binary-point position which avoids overflows. Then, the data word-lengths are determined to obtain a complete fixed-point specification. The data types which minimise the code execution time and respect the accuracy constraint are selected. Finally, the scaling operation locations are optimised to minimise the code execution time as long as the accuracy constraint is fulfilled. This conversion process is achieved under an accuracy constraint to obtain a fixed-point specification which satisfies the application performances. Thus, the computation accuracy must be evaluated and the accuracy constraint must be determined from application performances.

3.1.1. Compilation infrastructure

The floating-point C source algorithm is transformed into an intermediate representation with the compiler front-end. This intermediate representation (IR) specifies the application with a control and data flow graph (CDFG). The tool uses the SUIF compiler front-end [15], and the CDFG is generated from SUIF's internal-representation abstract trees. This CDFG is made up of different control flow graphs (CFGs) and data flow graphs (DFGs). Each CFG represents one of the application control structures. These structures correspond to basic blocks, conditional and repetitive structures. The core of conditional and repetitive structures is specified with a CFG. Each control structure block contains a specification of its input and output data. The basic block represents a set of sequential computations without control structure. The different computations of a basic block core which correspond to the signal processing part are represented with a data flow graph (DFG). The DFG includes the delay operations. To illustrate this intermediate representation, an FIR (finite impulse response) filter example is under consideration. The floating-point C source code is given in Algorithm 1 and the corresponding intermediate representation is presented in Figure 2.

The code generation is achieved with the flexible code generation tool CALIFE presented in [16] and the processor is described with the ARMOR language [17].

3.1.2. Fixed-point format

A fixed-point data is made up of an integer part and a fractional part as presented in Figure 3. The fixed-point format of a data is specified as (b, m, n), where b is the data word-length. The terms m and n are the binary-point positions referenced, respectively, from the most significant bit (MSB) and the least significant bit (LSB). In fixed-point arithmetic, m and n are fixed and lead to an implicit scale factor which stays constant during the processing.

A binary-point position is assigned to each o_i operation inputs and output $(m_{x'}, m_{y'}, m_{z'})$ as presented in Figure 4. In the same way, a word-length $(b_{x'}, b_{y'}, b_{z'})$ is assigned to each o_i operation operand. Let $\mathbf{b}_i = (b_{x'}, b_{y'}, b_{z'})$ and $\mathbf{m}_i = (m_{x'}, m_{y'}, m_{z'})$ be, respectively, the word-lengths

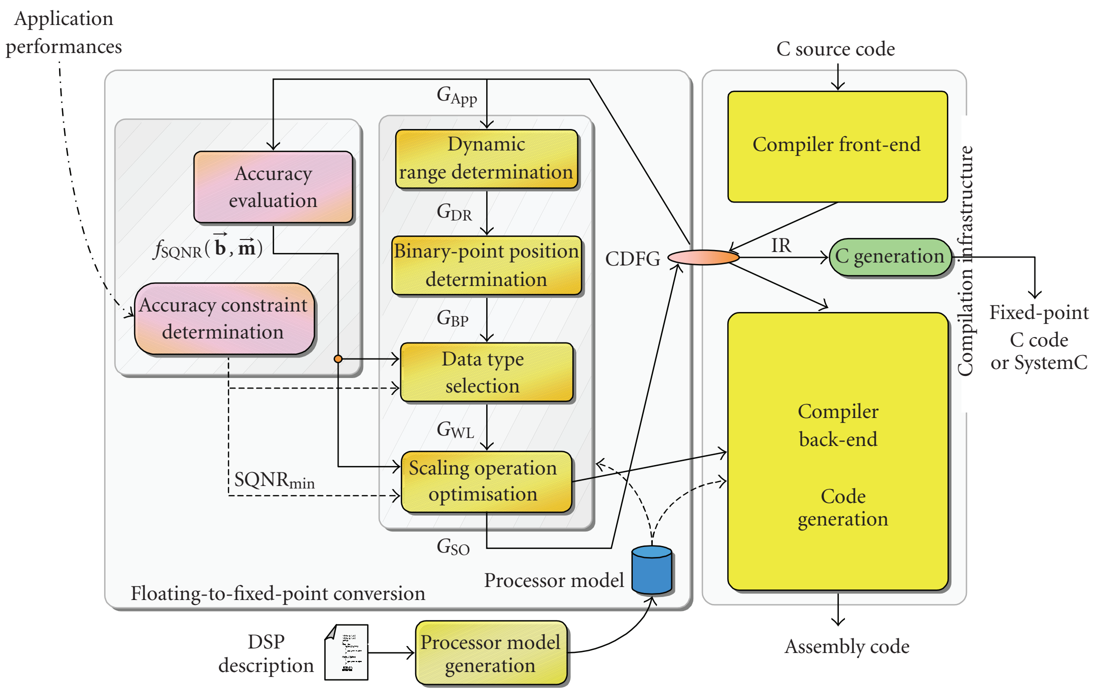

Application performances
C source code
G_{App}
Accuracy evaluation
Dynamic range determination
G_{DR}
Compiler front-end
$f_{SQNR}(\vec{b}, \vec{m})$
Binary-point position determination
CDFG
IR
C generation
Compilation infrastructure
Fixed-point C code or SystemC
Accuracy constraint determination
G_{BP}
Data type selection
G_{WL}
Compiler back-end
Code generation
$SQNR_{min}$
Scaling operation optimisation
G_{SO}
Processor model
Floating-to-fixed-point conversion
DSP description
Processor model generation
Assembly code

Figure 1: Methodology flow for the floating-to-fixed-point conversion. The tool is made up of two main blocks corresponding to the compilation infrastructure and to the floating-to-fixed-point conversion.

```
float h[32] = {−0.0297, . . . ,
0.897, 0.98, 0.897, . . . , −0.0297};
float x[32];
float y, acc;
float fir (float input)
{
int i;
   x[0] = input;
   acc = x[0] ∗ h[0] ;
   for (i = 31; i > 0; i − −)
   {
    acc = acc + x[i] ∗ h[i];
     x[i] = x[i − 1];
   }
   y = acc;
   return y;
}
```

Algorithm 1: Specification of the 32-tap FIR filter with the floating-point C source code.

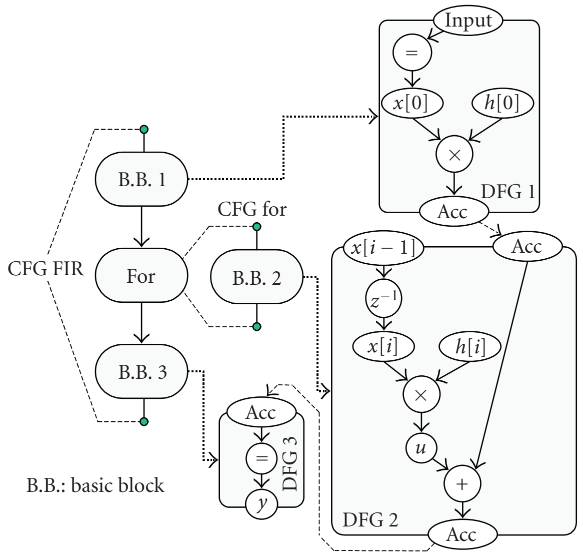


Figure 2: The control and data flow graph equivalent to Algorithm 1 (the node z^{-1} corresponds to a delay operation).

and the binary-point positions associated with the operation o_i. For a CDFG made up of N_o operations, $\vec{\mathbf{b}} = [\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_i, \ldots, \mathbf{b}_{N_o}]$ and $\vec{\mathbf{m}} = [\mathbf{m}_1, \mathbf{m}_2, \ldots, \mathbf{m}_i, \ldots, \mathbf{m}_{N_o}]$ are the vectors specifying, respectively, the word-length and the binary-point position of all CDFG operation operands.

3.1.3. *Computation accuracy management*

In our methodology, the SQNR metric is used to ensure that the fixed-point implementation verifies the application quality criteria. Thus, the accuracy constraint and the SQNR expression can be obtained from the application as explained in the following sections.

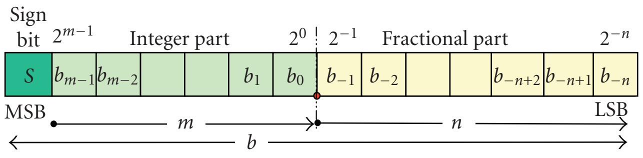


FIGURE 3: Fixed-point data specification: b, m, and n represent, respectively, the data word-length, the binary-point position referenced from the MSB (integer part), and the binary-point position referenced from the LSB (fractional part).

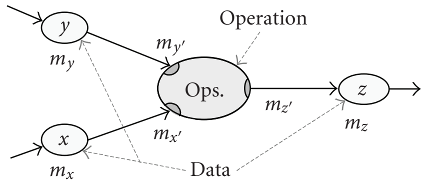


FIGURE 4: Binary-point position model for an operation. The binary-point position for the operation inputs and output are specified by $m_{x'}$, $m_{y'}$, and $m_{z'}$.

Accuracy constraint determination

The accuracy constraint corresponding to the minimal value ($\mathrm{SQNR}_{\min}$) of the SQNR is determined according to the application performance constraints. This SQNR minimal value is obtained with a floating-point simulation of the application as presented in Figure 5. The error due to the fixed-point conversion is modelled by a noise source (q_y) located at the system output. The power of this noise source is increased as long as the application performance constraints are respected. The SQNR constraint is determined from the maximal value of the noise source power which ensures that the application performances are still reached.

Computation accuracy evaluation

To determine the SQNR expression, the main challenge corresponds to the computation of the system output quantisation power. In fixed-point system, a quantisation noise q_{g_k} is generated when some bits are eliminated during a cast operation. Each quantisation noise source q_{g_k} is propagated inside the system and contributes to the output quantisation noise q_y through the gain α_k as presented in Figure 6. The goal of the analytical approach is to define the power expression of the output noise q_y according to the q_{g_k} noise source statistical parameters and the gains α_k between the output and the different noise sources.

For linear time-invariant systems, each α_k term is obtained from the transfer function between the system output and the q_{g_k} noise source. The transfer functions are determined from the data flow graph [18] representing the application [14]. They are obtained from the $\mathcal{Z}$ transform of the recurrent equations representing the system. The recurrent equations are built by traversing the graph from the inputs to the output. This technique requires that the graph be acyclic.

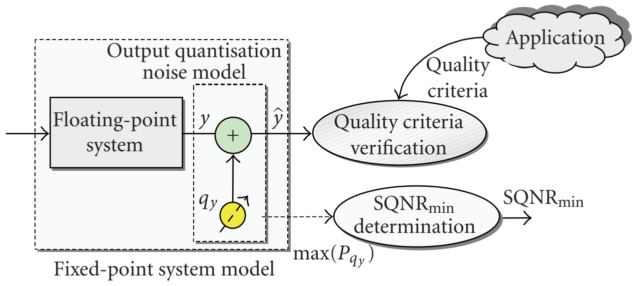


FIGURE 5: Technique to determine the accuracy constraint. The global error due to the fixed-point conversion is modelled by a noise source (q_y).

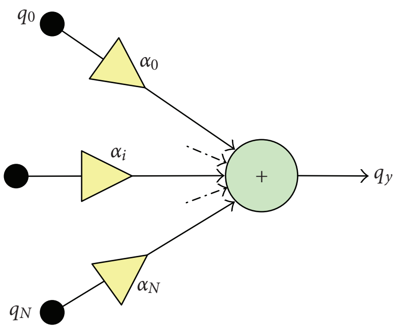


FIGURE 6: Output quantisation noise model in a fixed-point system. The system output noise q_y is a weighted sum of the different noise sources q_{g_k}.

Thus, the DFG is transformed into several directed acyclic graphs (DAGs) when cycles are present like in the case of recursive[1] structures.

In nonrecursive[2] and nonlinear systems, each α_k term is obtained from the signals associated with each operation involved in the q_{g_k} noise source propagation towards the output [19]. The α_k term expressions are built by traversing the acyclic graph from the inputs to the output. The statistical parameters of α_k are determined with a single floating-point simulation.

The q_{g_k} noise source statistical parameters are determined from the models presented in [20]. The statistical parameters depend on the number of bits eliminated and the data format after the cast operation. As described in (1), the SQNR is a function of the vector $\vec{\mathbf{b}}$ and $\vec{\mathbf{m}}$ specified in Section 3.1.2. This function is determined automatically from the data flow graph representing the application with the technique summarised in the previous paragraph and detailed in [14, 19]:

$$\mathrm{SQNR} = f_{\mathrm{SQNR}}(\vec{\mathbf{b}}, \vec{\mathbf{m}}). \tag{1}$$

[1] In a recursive structure, the system output depends on the input samples and the previous output samples.

[2] In a nonrecursive structure, the system output depends only on the input samples.

3.1.4. *Floating-to-fixed-point conversion*

For the floating-to-fixed-point conversion process, the data dynamic range is first evaluated. The results are used to determine the binary-point position of each data. Then, the data word-length is selected according to the data types supported by the targeted DSP. Finally, the fixed-point specification is optimised by moving the scaling operations to reduce the code execution time. The data word-length and the scaling operation location are optimised under accuracy constraint. These different transformations in the conversion process lead to the CDFG G_{DR}, G_{BP}, G_{WL}, and G_{SO} and are detailed in the following sections. The optimised fixed-point specification obtained after the conversion process can be transformed into a fixed-point C code or a SystemC code. This code can be used to simulate the fixed-point specification and to verify that the application quality criteria are respected.

3.2. *Data dynamic range determination*

The first stage of the methodology corresponds to the data dynamic range evaluation. This stage only depends on the application and the input signals. To evaluate an application data dynamic range, two approaches based on statistical or analytical methods can be used. The dynamic range can be computed from the data statistical parameters which are obtained with a floating-point simulation. The estimation results depend on the data used for the simulation. This approach produces an accurate estimation of the dynamic range from signal characteristics. It guarantees a low overflow probability for signals with the same characteristics. Nevertheless, overflows can occur for signals with different statistical properties.

The second class of methods corresponds to the analytical approaches which are based on the computation of the data dynamic range expressions from the input dynamic range. These methods guarantee that no overflow will occur but lead to a more conservative estimation. Indeed, the dynamic range expression is computed in the worst case. The data dynamic range can be obtained with the interval arithmetic theory [21]. The operation's output dynamic range is determined from its input dynamic. A worst-case dynamic range propagation rule is defined for each type of operation. Each data dynamic range is obtained with the help of the propagation rules during the application graph traversal. Thus, this technique cannot be used in the case of cyclic graphs like in recursive structures.

For linear time-invariant systems, the data dynamic range can be computed from the L1 or Chebyshev norm [22] according to the input signal frequency characteristics. These norms compute the data dynamic range in the case of linear time-invariant systems based on a nonrecursive or recursive structure. To evaluate the dynamic range of a data d_i from the system input x, the transfer function of the subsystem with the d_i output and the x input has to be determined.

In our methodology, these two analytical approaches have been combined to determine the data dynamic range in nonrecursive systems and in recursive linear time-invariant systems. The structure of this module is presented in Figure 7. The module input is the intermediate representation corresponding to the application CDFG G_{App}. The first step eliminates the control structures of the CDFG to obtain a data flow graph (DFG). For repetitive structures, the loops are unrolled, and for conditional structures, the branch which leads to the worst case is retained.

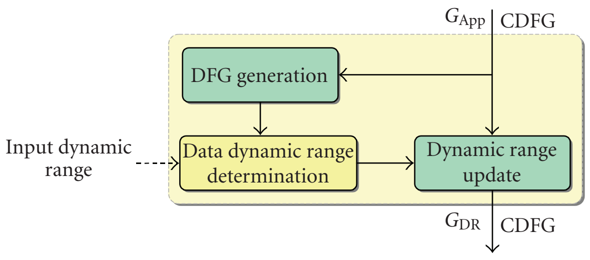


FIGURE 7: Methodology flow for the data dynamic range determination. The dynamic range is computed on the DFG representing the application and then the global CDFG is annotated with the dynamic range information.

The second step corresponds to the dynamic range computation for each data of the application DFG. For nonrecursive structures the dynamic range information are obtained by traversing the graph from the sources to the sinks. For each operation, a propagation rule is applied as defined in [21]. For recursive linear time-invariant structures, the transfer functions between the critical data and the inputs are determined with the technique presented in [14]. These critical data correspond to the output of the addition or subtraction operations. Then, the dynamic range is computed from the input dynamic range with the L1 or Chebyshev norm. For all other data, the dynamic range is obtained with the propagation rule technique.

The last step annotates the CDFG G_{App} data with the dynamic range and leads to the CDFG G_{DR}. For a data with only one instantiation in the CDFG, its dynamic range is equal to the dynamic range of the equivalent data in the application DFG. For data defined as vector (i.e., array) and used in loop, the vector dynamic range in the CDFG corresponds to the greatest value of the different vector elements used in the DFG. The dynamic range determination is more complex in the case of data with multiple instantiations like in the FFT (fast Fourier transform) butterfly where the butterfly inputs and outputs are stored in the same variables. The output vector dynamic range is multiplied by a factor of two at each FFT stage. Thus, the fixed-point format of the output vector must evolve at each stage. The first and the final values of the vector X dynamic range are specified through the input and the output loop structure and the evolution of the vector X dynamic range is specified through the input and the output CFG block which represents the ith FFT stage. This is illustrated in Figure 8. Consequently, the expression of the dynamic range evolution for a multiple instantiation data is determined from the different dynamic range values.

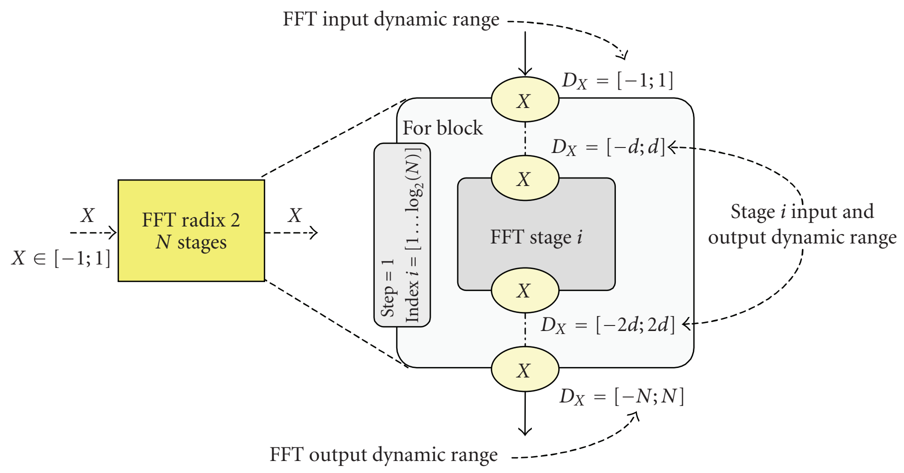


Figure 8: Specification of the data dynamic range for the FFT algorithm. The vector X dynamic range is specified for the FOR block input and output and for the FFT stage input and output.

3.3. *Binary-point position determination*

The second stage of the methodology corresponds to the determination of the data binary-point position. The dynamic range results are used to determine, for each data, the binary-point position which minimises the integer part word-length and avoids overflows. The architecture must be taken into account to determine the binary-point position. Indeed, many DSPs offer accumulator guard bits to manage the supplementary bits due to accumulations. Most of the DSPs achieve a MAC (multiply-accumulate) operation without loss of information. The adder and the multiplier output word-length is equal to the sum of the multiplier input word-lengths. Nevertheless, the dynamic range increase, due to successive accumulations, can lead to an overflow. Thus, many DSPs [23, 24] extend the accumulator word-length by providing guard bits. These supplementary bits ensure the storage of additional bits generated during successive accumulations. To avoid the introduction of costly scaling operations, these guard bits must be taken into account to determine the binary-point position.

The aim of this methodology stage is to obtain a correct fixed-point specification which guarantees no overflow. Moreover, this transformation must respect the different fixed-point arithmetic rules. Thus, scaling operations are included in the application to adapt the fixed-point format of a data to its dynamic range or to align the binary-point of the addition inputs. The input of this transformation is the CDFG G_{DR} where all the data are annotated with their dynamic range. The output is the CDFG G_{BP} where all the data are annotated with their binary-point position. A hierarchical approach is used to determine the data binary-point position. First, all the application DFGs are independently processed and then a global processing is applied to the CDFG to obtain a coherent fixed-point specification.

To determine the binary-point position (m) of each data, the different DFGs are traversed from the sources towards the sinks. For each data and operation, a rule is applied to obtain the binary-point position. This technique can be applied only on directed acyclic graph (DAG). Thus, the graph representing a DFG is firstly dismantled into a DAG if it contains cycles.

For a data x, the binary-point position m_x is obtained from the dynamic range with the following relation:

$$m_x = \left\lceil \log_2 \left(\max_n \left(|x(n)| \right) \right) \right\rceil. \tag{2}$$

A binary-point position is assigned to each operation input and output $(m_{x'}, m_{y'}, m_{z'})$ as presented in Figure 4. A propagation rule has been defined for each type of operation. These rules determine the value of $m_{x'}, m_{y'}, m_{z'}$ according to the binary-point position of the operation input and output data (m_x, m_y, m_z).

In the case of the multiplication, the binary-point positions of the inputs $(m_{x'}, m_{y'})$ correspond to those of the operation input data (m_x, m_y). The binary-point position of the multiplier output is directly obtained from the binary-point position of the operation inputs. Thus, the multiplier propagation rules are given by the following expressions:

$$\begin{aligned} m_{x'} &= m_x, \\ m_{y'} &= m_y, \\ m_{z'} &= m_{x'} + m_{y'} + 1. \end{aligned} \tag{3}$$

For addition and subtraction operations, a binary point position which is common to the operation inputs has to be defined to align the operation input binary point. This common position must guarantee no overflow. The lack of accumulator guard bits to store the supplementary bits due to overflow must be taken into account to determine the common binary-point position. Thus, to avoid overflow the common binary-point position m_c must be valid for the output

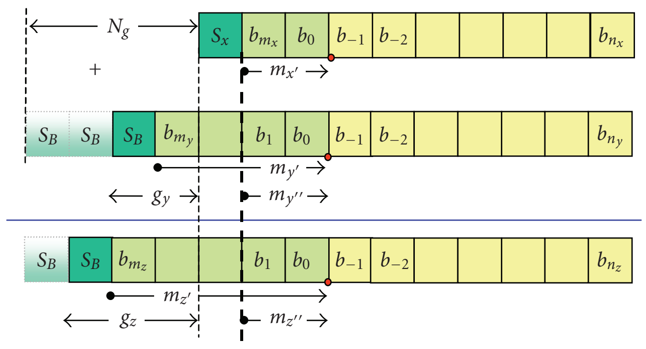

N_g
+
S_x b_{m_x} b_0 b_{-1} b_{-2} b_{n_x}
$m_{x'}$
S_B S_B S_B b_{m_y} b_1 b_0 b_{-1} b_{-2} b_{n_y}
$m_{y'}$
g_y
$m_{y''}$
S_B S_B b_{m_z} b_1 b_0 b_{-1} b_{-2} b_{n_z}
$m_{z'}$
g_z
$m_{z''}$

FIGURE 9: Binary-point position for an addition with N_g guard bits. The parameter g defines the number of guard bits used by the data.

data z and is defined as follows:

$$\begin{aligned} m_c &= \max(m_x, m_y, m_z), \\ m_{x'} &= m_c, \\ m_{y'} &= m_c, \\ m_{z'} &= m_c. \end{aligned} \tag{4}$$

If there are accumulator guard bits, the input and output word-lengths are different. Then, a common reference has to be defined to compare the binary-point positions. New binary-point positions ($m_{x''}$, $m_{y''}$, $m_{z''}$) referenced from the most significant bit of the data with the minimum word-length are computed for the inputs and the output as illustrated in Figure 9. A new parameter g corresponding to the number of guard bits used by the data is introduced as follows:

$$\begin{aligned} m_{x''} &= m_{x'} - g_x, \\ m_{y''} &= m_{y'} - g_y, \\ m_{z''} &= m_{z'} - g_z. \end{aligned} \tag{5}$$

Considering that the parameter g_z is unknown to determine m_c, it is fixed to N_g, which is the number of guard bits available for the accumulator:

$$m_c = \max(m_x - g_x, m_y - g_y, m_z - N_g). \tag{6}$$

The real number of guard bits used by the adder output is equal to

$$\begin{aligned} g_z &= m_z - m_c \quad \text{if } m_z > m_c, \\ g_z &= 0 \quad \text{if } m_z \le m_c; \end{aligned} \tag{7}$$

and, the binary-point positions of the adder inputs and output are equal to

$$\begin{aligned} m_{x'} &= m_c + g_x, \\ m_{y'} &= m_c + g_y, \\ m_{z'} &= m_c + g_z. \end{aligned} \tag{8}$$

The scaling operations required to obtain a correct fixed-point specification are inserted in the CDFG. For each operation, as represented in Figure 4, a scaling operation is introduced if the binary-point position of the data m_x (or m_y) is different from the binary-point position of the operation input $m_{x'}$ (or $m_{y'}$). For the operation output, a scaling operation is introduced if the binary-point positions $m_{z'}$ and m_z are different.

The results obtained for the FIR filter example presented in Figure 2 are given in Figure 10. The DFG associated with the second basic block (B.B. 2) of the FIR filter is presented. A processor with an accumulator without guard bit is considered. The data are annotated by their dynamic range and their binary-point position. For the operation, the output binary-point position is determined. A scaling operation must be introduced between the multiplication and the addition to align the binary-point position before the addition.

3.4. Data type selection

In the floating-to-fixed-point conversion process, each data type (word-length) is determined to obtain a complete fixed-point format for each CDFG data. This process must explore the diversity of the data types available in recent DSPs. Different elements of the data-path influence the computation accuracy as described in [4]. The most important element is the data word-length. Each processor is defined by its native data word-length which is the word-length of the data that the processor buses and data-path can manipulate in a single instruction cycle [25]. For most of the fixed-point DSPs, the native data word-length is equal to 16 bits. For ASIP (application-specific instruction-set processor) or some DSP cores like the CEVA-X and the CEVA-Palm [26], this native data word-length is customisable to adapt the architecture to the targeted applications. The computation accuracy is directly linked to the word-length of the data which are manipulated by the operations and depends on the type of instructions which are used to implement the operation.

Many DSPs support extended-precision arithmetic to increase the computation accuracy. In this case, the data are stored in memory with a greater precision. The data word-length is a multiple of the natural data word-lengths. Considering that extended-precision operations manipulate greater data word-lengths, an extended-precision operation is achieved with several single-precision operations. Consequently, this operation execution time is greater than the one of a single-precision operation.

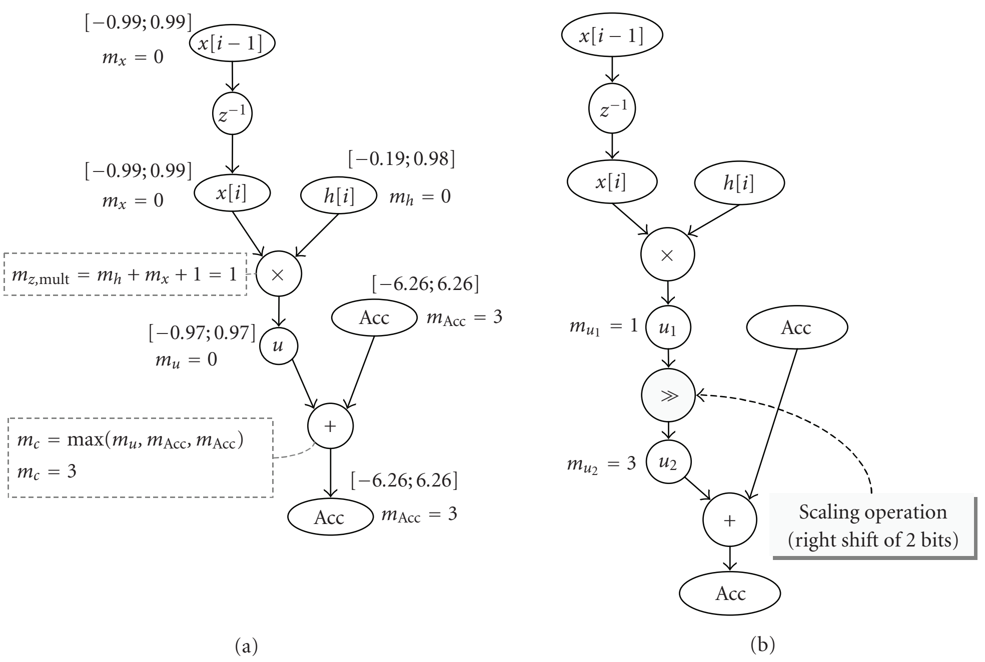


FIGURE 10: DFG representing the second basic block (B.B. 2) of the FIR filter specified in Figure 2. (a) The data dynamic range and the binary-point position for the DFG2 are specified. (b) DFG2 after the insertion of the scaling operation is shown. u, u_1, and u_2 are intermediate variables.

TABLE 1: Word-length of the data which can be manipulated by different DSPs offering SWP capabilities for arithmetic operations.

Processor	Data types (bits)
TMS320C64x (T.I.) [29]	8, 16, 32, 40, 64
TigerSHARC (A.D.) [28]	8, 16, 32, 64
SP5, UniPhy (3DSP) [30]	8, 16, 24, 32, 48
CEVA-X1620 (CEVA) [31]	8, 16, 32, 40
ZSP500 (LSI Logic) [32]	16, 32, 40, 64
OneDSP (Siroyan)	8, 16, 32, 44, 88

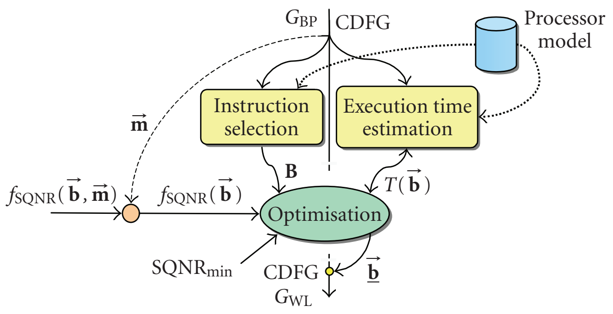


FIGURE 11: Flow of the data type selection process. This optimisation process uses the SQNR expression f_{SQNR} to evaluate the computation accuracy. It requires selecting the instructions for each operation and to evaluate the code execution time T. The data of the output CDFG G_{WL} are annotated with their optimised word-length specified through the vector $\underline{\mathbf{b}}$.

To reduce the code execution time, some recent DSPs can exploit the data-level parallelism by providing SWP (subword parallelism) capabilities. An operator (multiplier, adder, shifter) of word-length N is split to execute k operations in parallel on subwords of word-length N/k. This technique can accelerate the code execution time up to a factor k. Thus, these processors can manipulate a wide diversity of data types as shown in Table 1 for several recent DSPs. In [27], this technique has been used to implement a CDMA (code-division multiple access) synchronisation loop into the TigerSharc DSP [28]. The SWP capabilities offer the opportunity to achieve an average 6.6 MAC per cycle with two MAC units.

The main goal of the code generation process is to minimise the code execution time under a given accuracy constraint. Thus, our methodology selects the instructions which respect the global accuracy constraint and minimise the code execution time. The methodology flow is presented in Figure 11. The input of this transformation is the CDFG G_{BP} where all the data are annotated with their binary-point position specified through the vector $\vec{\mathbf{m}}$. The output is the CDFG G_{WL} where all the data are annotated with their optimised word-length specified through the vector $\underline{\vec{\mathbf{b}}}$. This transformation leads to a complete fixed-point specification. This optimisation process use the SQNR expression $f_{\text{SQNR}}(\vec{\mathbf{b}}, \vec{\mathbf{m}})$ to evaluate the computation accuracy. Before starting the optimisation process, for each operation, the different instructions which can be used are selected. During the optimisation process, the application execution time is estimated.

3.4.1. Code execution time estimation

The processor is modelled by a data flow instruction set. These instructions implement arithmetic operations. The instructions are obtained from one or several instructions of the processor instruction set. Each data flow instruction j_k is characterised by its function γ_k, its operand word-length b_k, and its execution time t_k. This execution time is obtained from the processor model. For SWP instructions, the execution time is set to the processor instruction execution time divided by the number of operations executed in parallel. For the extended-precision instructions, the execution time is the sum of the execution time of the processor instructions used to implement this operation. A processor model example is presented in Figure 12(a).

The global application execution time is estimated from the instructions selected for the N_o operations of the CDFG. Nevertheless, the goal is not to obtain an exact execution time estimation but to compare two instruction lists and to select the one that leads to the minimal execution time. Thus, a simple estimation model is used to evaluate the execution time $T(\vec{\mathbf{b}})$ of the CDFG. This time depends on the type of instruction used to execute the CDFG operations and thus T is a function of the vector $\vec{\mathbf{b}}$ which specifies the word-length of the CDFG operation operands. The time $T(\vec{\mathbf{b}})$ is estimated from the execution time t_i and the number of executions n_i of each o_i operation as follows:

$$T(\vec{\mathbf{b}}) = \sum_{i=1}^{N_o} t_i \cdot n_i. \tag{9}$$

This estimation method is based on the sum of the instruction execution times and leads to accurate results for DSPs without instruction parallelism. For DSPs with instruction-level parallelism (ILP), this method does not take account of the instructions executed in parallel. Nevertheless, this estimation can be used to compare adequately two instruction lists in the case of a processor with ILP.

For single-precision and SWP instructions, the gains due to the transformation (code parallelisation) of the vertical code into a horizontal one are similar. Indeed, the two instruction lists use the same functional units at the same clock cycles. The difference lies in the functionality of the processor unit. For SWP instructions, the functional units manipulate fractions of a word instead of the entire word. Thus, the gains due to the code parallelisation are identical with SWP and single-precision instructions.

An extended-precision instruction is achieved with several single-precision instructions. Thus, in the best case and after the scheduling stage, the extended-precision instruction execution time can be equal to the execution time of the single-precision instructions. In this case, the single-precision instructions must be favoured if the precision constraint is fulfilled to reduce the data memory size. Therefore, the extended-precision instruction execution time is set to the maximal value to select them only if the single-precision instructions cannot fulfil the precision constraint.

This approach for the code execution time estimation can be improved with more accurate techniques such as those presented in [33, 34]. On the other hand, the optimisation time will be increased.

3.4.2. Data type selection

In this section, the data type selection process is described. For each CDFG operation o_i, the different instructions, achieving o_i, are selected. Let I_i be the set specifying the instructions selected for the operation o_i. Let $\mathbf{B}_i$ be the set specifying all the possible word-lengths for the o_i operation operands. Thus, for each operation o_i, the optimised word-length $\hat{b}_i$ ($\hat{b}_i \in \mathbf{B}_i$), that is, which minimises the global execution time $T(\vec{\mathbf{b}})$ and respects the minimal precision constraint, must be selected. Consequently, the application execution time $T(\vec{\mathbf{b}})$ is minimised as long as the accuracy constraint ($\mathrm{SQNR}_{\min}$) is fulfilled as described with the following equation:

$$\min_{\vec{\mathbf{b}} \in \mathbf{B}} \left(T(\vec{\mathbf{b}})\right) \quad \text{subject to } f_{\mathrm{SQNR}}(\vec{\mathbf{b}}) \geq \mathrm{SQNR}_{\min}. \tag{10}$$

Considering that the number of values for each variable b_i is limited, the optimisation problem can be modelled with a tree. This optimisation process is illustrated with an FIR filter example in Figure 12. To obtain the optimal solution, the tree must be explored exhaustively. This technique leads to an exponential optimisation time. To explore efficiently this tree a *branch-and-bound* algorithm is used with four techniques to limit the search space. These techniques are presented in the next section.

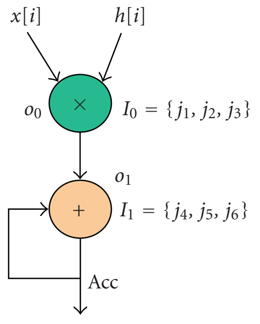
3.4.3. Search space limitation

The tree modelling of this optimisation problem offers the capability to exhaustively enumerate solutions. Nevertheless, all the instruction combinations are not valid. Let us consider two operations o_l and o_k where the o_l operation input is the o_k operation result. In this case, the number of bits n_l^{in} for the o_l input fractional part cannot be strictly greater than the number of bits n_k^{out} for the o_k output fractional part. Thus, the instruction tested for the operation o_l is valid if $n_k^{\mathrm{out}} \geq n_l^{\mathrm{in}}$. If this condition is not respected, the exploration of the subtree is stopped and a new instruction is tested for the operation o_l. This technique reduces significantly the search space.

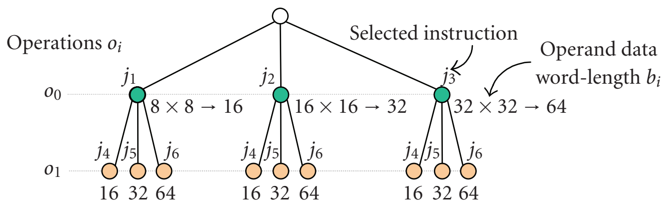
In the *branch-and-bound* algorithm, the partial solutions are evaluated to stop the tree exploration if they cannot lead to the best solution. At the tree level l, the exploration of the subtree induced by the node representing b_l can be stopped if the minimal execution time which can be obtained during the exploration of this subtree is greater than the minimal execution time which has already been obtained. Considering that only the word-lengths b_0 to b_l are already defined, the minimal execution time is determined by selecting for the operation o_j ($j \in [l+1, N_o]$) the instruction with the minimal execution time t_j.

Instruction j_k	Function y_k	Execution time t_k	I/O operand word-length		
			$\mathbf{b}^{\text{in}\,1}$	$\mathbf{b}^{\text{in}\,2}$	$\mathbf{b}^{\text{out}}$
j_1	MULT	0.25	8	8	16
j_2	MULT	0.5	16	16	32
j_3	MULT	1	32	32	64
j_4	ADD	0.25	16	16	16
j_5	ADD	0.5	32	32	32
j_6	ADD	1	64	64	64

(a)



(b)



(c)

FIGURE 12: Data word-length optimisation process for an FIR filter. (a) Model of the processor data flow instruction set. (b) FIR filter data flow graph. (c) Model with a tree of the different solutions for the optimisation.

At the tree level l, the exploration of the subtree induced by the node representing b_l can be stopped if the maximal SQNR which can be obtained during the exploration of this subtree is lower than the precision constraint (SQNR$_{\text{min}}$). The SQNR maximal value is obtained by fixing the word-lengths b_j ($j \in [l+1, N_o]$) to their maximal value. Indeed, considering that the SQNR is a monotonic and nondecreasing function, the SQNR maximal value is obtained for the maximal operand word-length.

This optimisation technique based on a tree traversal is sensitive to the node evaluation order. To find quickly a good solution to reduce the search space, the variables with the most influence on the optimisation process must be evaluated first. The variables are sorted by their influence on the global execution time. The influence of the operation o_i on the execution time is obtained from the number of times (n_i) that this o_i operation is executed.

For applications with a great number of variables, the optimisation time can become important. To obtain reasonable optimisation time, the optimisation is achieved in two steps. Firstly, the variables corresponding to the data word-length are considered as positive real numbers and a constrained nonlinear optimisation technique is used to minimise the code execution time under accuracy constraint. The optimisation technique is based on the sequential quadratic programming (SQP) [35]. Let $\widetilde{b}_i$ be the optimised solution obtained with this technique for the variable b_i. Secondly, the technique based on the *branch-and-bound* algorithm presented previously is applied with a reduced number of values per variable. For each variable b_i, only the values which are members of B_i and immediately higher and lower than $\widetilde{b}_i$ are retained. Thus only two values are tested for each variable and the search space is dramatically reduced.

An optimisation time less than 200 *seconds* has been obtained for the *branch-and-bound* algorithm with 35 variables and four alternatives per variable. In this case, only the two first techniques corresponding to the instruction combination restriction and the partial solution evaluation were used. For the same application, this optimisation time is dramatically reduced when two alternatives are tested for each variable like for the last search space reduction technique which achieves the optimisation in two steps.

3.5. *Scaling operation optimisation*

The previous methodology stages, that correspond to the determination of the data word-length and the binary-point position, lead to an optimised fixed-point specification in terms of accuracy. Indeed, scaling operations have been inserted to maintain a sufficient computation accuracy. These scaling operations are used to adapt the fixed-point format to the data dynamic range or to insert additional bits in the integer part to avoid overflows. Nevertheless, these scaling operations increase the code execution time. The aim of this part is to optimise the fixed-point data formats to minimise the code execution time $T(\vec{\mathbf{m}})$ as long as the accuracy constraint is fulfilled. The execution time is reduced by moving the scaling operations. These scaling operation transfers modify the data binary-point position specified through the vector $\vec{\mathbf{m}}$. Thus, this optimisation problem can be expressed as follows:

$$\min_{\vec{\mathbf{m}}} (T(\vec{\mathbf{m}})) \quad \text{subject to } f_{\text{SQNR}}(\vec{\mathbf{m}}) \geq \text{SQNR}_{\min}. \tag{11}$$

3.5.1. *Scaling operation transfers*

Scaling operations based on a left-shift adapt the fixed-point format to the data dynamic range. The number of bits m used for the integer part is reduced, because this one is too high compared to the data dynamic range. This bit number reduction for the integer part can be delayed. Thus, this scaling operation achieved with a left shift can be moved towards the application graph sinks.

Scaling operations based on a right shift realise the insertion of supplementary bits for the integer part to support the data dynamic range increase. This supplementary bits insertion can be brought forward. Thus, this scaling operation achieved with a right shift can be moved towards the application graph sources. Nevertheless, left-shift operations are inserted after a set of accumulations which use guard bits. This operation ensures the guard bit recovering before spilling the data in memory. In this case, the binary-point position is not changed. Consequently, this operation must not be moved, otherwise the guard bits would be lost.

To move the scaling operations, a propagation rule is defined for each class of operations. When a right shift is moved towards a multiplication operation, one of the inputs must be selected to receive the scaling operation. In the case of linear systems, two alternatives are available to move a right shift. These scaling operations can be moved towards the system inputs or towards the coefficients. For this last case the degradation of the SQNR is less important. But in the case of linear filters, the degradation of the frequency response due to the coefficient quantisation is more significant.

3.5.2. *Architecture influence on the scaling operation cost*

Different classes of shift registers are available in DSPs to scale the data. In some processors [24, 36], a specialised shift register is located at the output or at the input of an operator and several specific shifts can be achieved. Thus, the operator input or output can be scaled without supplementary cycle.

For more flexibility, most of the recent DSPs offer a barrel shifter which is able to perform any shift operation in one cycle. In traditional DSPs [23, 24, 36] based on a MAC (multiply-accumulate) structure, the registers are dedicated to a specific operator. The barrel shifter is connected to the accumulation register and can only scale efficiently the output of an addition. To analyse the additional cost due to the scaling operation, several experiments have been conducted on the DSPStone benchmark [37]. Different locations of a scaling operation in the applications have been tested. This scaling operation requires between one and five cycles for the TMS320C54x [23] and between one and four cycles for the OakDSPCore [38]. These additional cycles required for the scaling operation are due to the transfer between the registers. The evaluation of the scaling operation execution time requires the knowledge of the data location before and after the shift instruction. Thus, the instruction list used to implement the scaling operation has to be determined. This list is obtained with the code selection stage.

In homogeneous architectures a register file is connected to a set of operators working in parallel like in VLIW (very long instruction word) DSPs [28, 29]. For these architectures, the barrel shifter can scale the input or the output of any operation in one cycle. For processors with instruction-level parallelism, the scaling operation cost depends on the opportunity to execute this operation in parallel with the other instructions. To illustrate and quantify this concept, the extra cost due to a scaling operation has been measured on the DSPStone benchmark implemented into the TMS320C64x VLIW DSP [4]. For these applications based on a MAC operation, the application execution times have been measured with and without a scaling operation executed after the multiply operation. Let T_{r_i} and $T_{\overline{r_i}}$ be the code execution times, respectively, with and without a scaling operation r_i. The extra cost $\mathcal{C}_r$ defined in (12) corresponds to the ratio between the additional execution time due to the scaling operation $(T_{r_i} - T_{\overline{r_i}})$ and the application execution time without this scaling operation $(T_{\overline{r_i}})$. This extra cost depends on the average IPC (instructions per cycle) obtained for the application without a scaling operation. When the IPC is closed to its maximal value, the extra cost can be relatively important (47%). Indeed, most of the functional units are used and supplementary cycles are required to execute the scaling operations. When the IPC decreases, the extra cost diminishes and can climb down to 0%. Thus, these results underline that the scaling operation execution time can be evaluated only during the scheduling stage:

$$\mathcal{C}_r = \frac{T_{r_i} - T_{\overline{r_i}}}{T_{\overline{r_i}}}. \tag{12}$$

To optimise the scaling operation location, two approaches have been defined according to the DSP architecture and more particularly the DSP instruction-level parallelism (ILP).

3.5.3. *DSPs without instruction-level parallelism*

In this part, the approach proposed for processors without instruction-level parallelism is explained. For this class of DSPs, only one instruction is executed per cycle and the parallelism is specified through complex instructions. The flow of the optimisation of the scaling operation location is presented in Figure 13. The input of this transformation is the CDFG G_{WL} where all the data are annotated with their optimised fixed-point specification. The output is the CDFG G_{SO} where the location of the scaling operations has been optimised. This optimisation process uses the SQNR expression $(f_{\text{SQNR}}(\vec{\mathbf{b}}, \vec{\mathbf{m}}))$ to evaluate the computation accuracy. The technique used to estimate the extra execution time due to scaling operations and the algorithm proposed to minimise this execution time are explained.

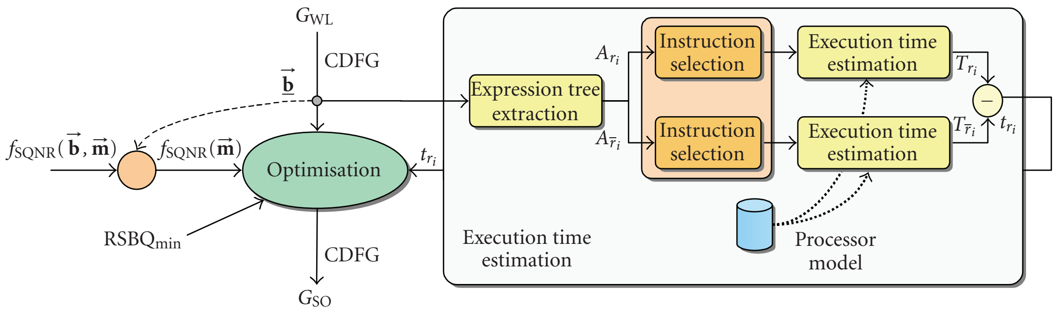


FIGURE 13: Flow to optimise the scaling operation location for DSP without instruction-level parallelism. The execution time t_{r_i} of the scaling operation r_i is estimated.

For a scaling operation r_i, let t_{r_i} be its execution time and n_{r_i} the number of times that r_i is executed. The scaling operation cost is defined as the product of r_i and n_{r_i}. For this class of DSP architectures, the global execution time of the N_{SO} scaling operations located in the application CDFG is determined with the following expression:

$$T_{SO} = \sum_{i=1}^{N_{SO}} n_{r_i} \cdot t_{r_i}. \tag{13}$$

The execution time t_{r_i} is equal to the difference between T_{r_i} and $T_{\overline{r_i}}$. The times T_{r_i} and $T_{\overline{r_i}}$ correspond to the code execution times, respectively, with and without the scaling operation r_i. The technique used to evaluate the times T_{r_i} and $T_{\overline{r_i}}$ is represented in the right part of Figure 13. First of all, the expression tree which includes the scaling operation r_i is extracted. Then a code selection is applied on this expression tree with (A_{r_i}) and without ($A_{\overline{r_i}}$) the scaling operation. The execution time is directly computed from the instruction list selected for the expression tree. It corresponds to the sum of the different instruction execution times and it leads to a sufficient accurate estimation of the code execution time for this class of DSP architectures. Indeed, the parallelism is specified through complex instructions and can be detected during the code selection stage. Nevertheless, this technique can be improved by taking account of the pipeline hazards with the technique proposed in [39]. The adjacent instructions can be analysed to determine if a pipeline hazard can occur.

The scaling operation optimisation problem is solved with an iterative algorithm. For each iteration, a scaling operation is moved and this transfer is validated if the accuracy constraint is respected. The scaling operations are processed by cost-decreasing order to consider costly operations first. After each transfer, the application accuracy is evaluated. If the accuracy constraint is no longer respected, the scaling operation is replaced in the location which leads to the minimal execution time and this operation will not be moved after. If the accuracy constraint is still fulfilled, the scaling operation transfer is validated. Then, the scaling operation costs are computed. In the next iteration, the scaling operation with the maximal cost is processed. The algorithm finishes when no scaling operation can be moved.

For the FIR filter example presented in Figure 2, the scaling operations have been optimised for the TMS320C50 architecture model. The scaling operations are moved towards the system input. The fixed-point C code generated before and after the optimisation process are presented in Algorithms 2 and 3, respectively. This optimisation process decreases the scaling operation execution time T_{SO} from 120 cycles to 0. Thus, the global code execution time is reduced by 36%. On the other hand, the output SQNR is reduced by 4.5 dB.

3.5.4. DSPs with instruction-level parallelism

For processors with instruction-level parallelism, the estimation of the execution time must be coupled with the scheduling stage to take account of the partial instructions which are executed in parallel. Indeed, the scaling operation cost depends on the opportunity to execute this operation in parallel with the other instructions. Thus, the goal of our approach is to find the scaling operation location which enables the execution of the shift operation in parallel with other instructions. The aim is to find the scheduling which minimises the increase of time compared to the scheduling obtained without the scaling operations.

For a scaling operation r_k located between the operations o_i and o_j, the scaling operation cost $c_{k,ij}$ is defined with the expression (14). The term η_{ij} defines the maximal number of scaling operations which can be inserted between the operations o_i and o_j without increasing the execution time compared to a solution without scaling operation. This term depends on the operations o_i and o_j mobility and the processor resource usage rate. This term is computed from the operation execution date obtained with a list scheduling algorithms in a direct and forward sense. For this, the operation o_i is executed as soon as possible and the operation o_j is executed as late as possible. When no scaling operation can be inserted, the term η_{ij} is null and the cost is equal to its maximal value:

$$c_{k,ij} = \frac{1}{1+\eta_{ij}}. \tag{14}$$

The scaling operation optimisation is achieved with an iterative process made up of three steps corresponding to

```
short h[32] = {−973, . . . , 29418, 32112,
29418, . . . , −973};
short x[32];
short y;
int acc;

short fir (short input)
{

int i;
    *x = input;
    acc = *x * *h ≫ 2;

    for (i = 31; i > 0; i − −)
    {
         acc = acc +x[i] * h[i] ≫ 2;
         x[i] = x[i − 1];
    }
    y = (short) (acc);

    return y;
}
```

Algorithm 2: Fixed-point C code for the FIR filter before the scaling operation optimisation.

```
short h[32] = {−973, . . . , 29418, 32112,
29418, . . . , −973};
short x[32];
short y;
int acc;

short fir (short input)
{

int i;
    *x = input ≫ 2;
    acc = *x * *h;

    for (i = 31; i > 0; i − −)
    {
         acc = acc +x[i] * h[i];
         x[i] = x[i − 1];
    }
    y = (short) (acc);

    return y;
}
```

Algorithm 3: Fixed-point C code for the FIR filter after the scaling operation optimisation.

the scaling operation cost computation, the transfer of some scaling operations, and the scheduling. The scaling operations are processed by cost-decreasing order. They are moved as long as their cost is equal to one and the accuracy constraint is fulfilled.

4. EXPERIMENTS AND RESULTS

4.1. *Floating-to-fixed-point conversion for a WCDMA receiver*

The aim of this part is to show the interest of our approach to obtain an optimised fixed-point specification in the case of a real-life application corresponding to a WCDMA receiver. Especially, this experiment underlines the benefits provided by the data type selection stage to reduce the code execution time.

4.1.1. *WCDMA receiver description*

The considered application corresponds to a receiver used in the base station for the third-generation telecommunication systems. UMTS (Universal Mobile Telecommunications System) is based on the wideband code-division multiple-access (WCDMA) norm [40]. The information data (DPDCH) and the control data (DPCCH) are spread with an orthogonal variable-spreading-factor code (OVSF), and then scrambled by a specific spreading sequence (Kasami codes).

In the receiver part, the complex received signal is made up of different delayed copies of the transmitted signal due to the multipaths inside the radio channel. The RAKE-receiver concept is based on the combination of the different multipath components to improve the quality of the decision on symbols. Each multipath signal is processed by a finger which correlates the received signal by a spreading code. The RAKE receiver and the different finger structures are detailed in Figure 14. The signal $y(k)$ corresponds to the combination of the different finger outputs $y_l(k)$. To combine the different finger results, the complex amplitude α_l of the lth path must be estimated and removed for each multipath. The symbols are decoded by multiplying the received signal with a synchronised version of the code generated in the receiver. The synchronisation between the code and the received signal is realised by a delay-locked loop (DLL).

For each finger, the symbols (DPDCH/DPCCH) are estimated with the symbol decoder structure presented in Figure 15. Thanks to the complex multiplication (CM 1) of the received signal by the conjugate of the Kasami code $c_K^*(n)$ the unscrambling operation is performed. Then, the phase distortion resulting from the transmission channel is removed with the complex multiplication (CM 2) with the conjugate of the complex amplitude estimation ($\hat{\alpha}_l^*$). At last, the despreading operation with OVSF code ($c_{\mathrm{OVSF}_I}(n)$ and $c_{\mathrm{OVSF}_Q}(n)$) transforms the wideband received signal into a narrowband signal. This operation decodes the transmitted symbols $y_l(k)$.

4.1.2. *Data type selection*

Recent DSPs like the TMS320C64x from Texas Instruments provide a wide diversity of data types with the SWP capabilities. The data type selection is a tradeoff between the computation accuracy and the code execution time. To illustrate the different opportunities offered by this class of architectures, the complex correlator used in the RAKE receiver has been implemented with different data types. For each solution S_i, the execution time and the signal-to-quantification-noise ratio (SQNR) metric are evaluated. The results are presented

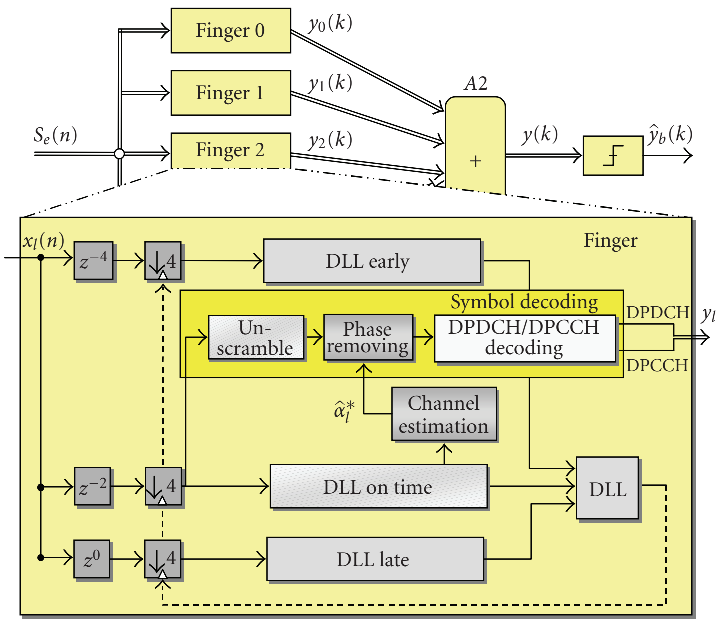


Figure 14: Schematic of the RAKE receiver and the finger for a base station. The RAKE receiver achieves the combination of the different finger results.

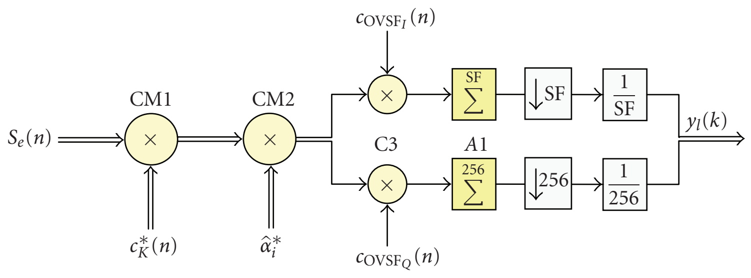


Figure 15: Symbol decoding subsystem for a base-station receiver.

in Table 2 and the word-lengths of the operation operands are reported. These different results have been obtained by using our methodology with different accuracy constraints ($\mathrm{SQNR}_{\min}$). The execution time ($T_{\mathrm{norm}}(\vec{\mathbf{b}})$) is normalised in relation to the execution time of a classical implementation based on single-precision instructions (multiplication: $16 \times 16 \Rightarrow 32$ bits; addition: $32 + 32 \Rightarrow 32$ bits).

Before determining the RAKE-receiver fixed-point specification, the accuracy constraint must be defined. This minimal value of the SQNR ($\mathrm{SQNR}_{\min}$) is defined according to the system performance constraints. In the case of the WCDMA receiver, the performances are specified through the maximal value of the bit error rate (BER). The accuracy constraint has been defined so that the system output BER is slightly modified after the fixed-point conversion process. Compared to the floating-point implementation, the maximal BER degradation due to fixed-point computation is fixed to 5%. The SQNR minimal value is obtained with a floating-point simulation with the technique explained in Section 3.1.3. For the WCDMA receiver, this accuracy constraint determination process leads to a minimal SQNR equal to 12.5 dB.

The WCDMA receiver fixed-point specification has been obtained with our methodology. The input data (receiving Nyquist filter output) word-length was fixed to 8 bits. The word-lengths of the main data for the symbol decoding subsystem of the RAKE receiver are summarised in Table 3. For this experiment, the Texas Instruments code generation tool is used to benefit from the high performance of the C compiler and more particularly the software pipelining technique. Thus, the C source code is modified to include the different data types from the fixed-point specification. Intrinsic functions are used to express the data parallelism. The data parallelisation must be achieved by the user to exploit the processor SWP capabilities.

To analyse the improvement due to the data type selection stage, the execution times of the code obtained with a classical implementation based on single-precision instructions ($T_{\mathrm{unopt}}(\vec{\mathbf{b}})$) and with our methodology ($T_{\mathrm{opt}}(\vec{\mathbf{b}})$) have been measured. In the classical approach, the data types are

Table 2: Results of the complex correlator implementation for different data types. T_{norm} is the execution time normalised in relation to the classical implementation (impl. 2) execution time.

S_i	T_{norm}	SQNR (dB)	Operand word-length (bits) Multiplication bits × bits ⇒ bits	Operand word-length (bits) Addition bits + bits ⇒ bits
1	0.6	51	$8 \times 8 \Rightarrow 16$	$16 + 16 \Rightarrow 16$
2	1	89	$16 \times 16 \Rightarrow 32$	$32 + 32 \Rightarrow 32$
3	1.55	151	$32 \times 16 \Rightarrow 32$	$32 + 32 \Rightarrow 32$
4	2.1	170	$32 \times 16 \Rightarrow 64$	$64 + 64 \Rightarrow 64$

Table 3: Data word-length for the symbol decoding subsystem of the RAKE receiver. The data and the operations are presented in Figure 15.

Operations		Data	Data type (bits)
		x_l	8
CM 1	MULT		$8 \times 8 \rightarrow 16$
	ADD		$16 + 16 \rightarrow 16$
CM 2	MULT		$16 \times 16 \rightarrow 32$
	ADD		$16 + 16 \rightarrow 16$
$M3$	MULT		$16 \times 16 \rightarrow 32$
$A1$	ADD		$16 + 16 \rightarrow 16$
$A2$	ADD		$16 + 16 \rightarrow 16$
		y_l	16

not optimised and thus only the single-precision instructions are used (multiplication: $16 \times 16 \Rightarrow 32$ bits; addition: $32+32 \Rightarrow 32$ bits). Given that the two floating-to-fixed-point conversion methods presented in Section 2.1 do not optimise the data type, the results obtained with these approaches correspond to the classical implementation. In our approach, the code is obtained from the fixed-point specification determined with our floating-to-fixed point conversion methodology. The accuracy constraint and the DSP architecture offer the opportunity to use the SWP instructions. To compare these two approaches, the ratio F between the two execution times $T_{\text{unopt}}(\vec{\mathbf{b}})$ and $T_{\text{opt}}(\vec{\mathbf{b}})$ is computed. This improvement factor F, defined in (15), corresponds to the acceleration factor due to the data type selection:

$$F = \frac{T_{\text{unopt}}(\vec{\mathbf{b}})}{T_{\text{opt}}(\vec{\mathbf{b}})}. \tag{15}$$

Different experiments have been achieved on the symbol decoding and the synchronisation subsystems for several values of the fingers number. The results, presented in Table 4, underline the benefit of the SWP instructions. Our approach reduces the code execution time by a factor between 1.91 and 3.51.

4.2. Optimisation of the scaling operation location

In this section, some experiments have been conducted to show our approach's interest to optimise the scaling operation location for DSP based on conventional architecture. These experiments are achieved with the C50 and the C54x DSPs from Texas Instruments. These two processors are based on a classical MAC structure. The C54x DSP is made up of an accumulator with eight guard bits and a barrel shifter connected to the accumulator register. The C50 offers no guard bits and the scaling capabilities based on specialised shift registers are limited. A *prescaler* register is available to shift the data which are loaded from memory and a *postscaler* register provides the capability to shift the data when they are stored in memory.

The different experiment results are given in Table 5. The scaling operation execution time T_{SO} is given before and after the optimisation of the scaling operation location to analyse the improvement due to the optimisation process. The execution time T_{SO} (number of cycles) corresponds to the application execution time difference with and without the scaling operations. The accuracy degradation Δ_{SQNR} (dB) due to the scaling operation transfers is measured.

The two first applications correspond to a finite impulse response and an infinite impulse response filters. The complex correlator achieves the correlation between a complex signal and a complex bipolar code made up of N points. The four last applications are used in the WCDMA receiver for third-generation telecommunication systems. These applications are described in the previous section. The receivers for the mobile terminal (MT) and for the base station (BS) are similar except for the location of the phase removing processing. In the base station the phase removing is achieved during the symbol decoding and in the mobile station the phase removing is achieved after the symbol decoding and before the output finger combination.

For the C54x, the guard bits ensure a fixed-point specification with a limited number of scaling operations. Except for the IIR filter, these scaling operations correspond to left shifts required to align the guard bits before storing in memory the data which was in the accumulator register. Thus, these scaling operations cannot be moved and the scaling operation optimisation does not reduce the scaling operation cost. In the IIR filter, the guard bits are not sufficient to limit the number of scaling operations. A scaling operation is required to adapt the format of the recursive and the nonrecursive part outputs. This scaling operation can be moved to reduce the scaling operation cost.

For the C50, the lack of guard bits leads to a fixed-point specification with a high execution time for the scaling

Table 4: SWP improvement factor F. This factor F corresponds to the acceleration factor due to the data type selection.

Number of fingers	Code execution time improvement factor F Symbol decoding subsystem	Code execution time improvement factor F Synchronisation subsystem
1	2.83	1.91
2	2.79	2.79
4	3.51	3.18

TABLE 5: Optimization of the scaling operation location for different applications implemented in the C54x and the C50 DSPs. The scaling operation execution time T_{SO} (number of cycles) and the SQNR degradation Δ_{SQNR} (dB) are evaluated.

Applications	TMS320C54x			TMS320C50		
	Initial	*After optimisation*		*Initial*	*After optimisation*	
	T_{SO}	Δ_{SQNR}	T_{SO}	T_{SO}	Δ_{SQNR}	T_{SO}
FIR 32-tap filter	1	0	1	128	−4.5	0
Second-order IIR filter	3	−8.6	2	7	−8.6	0
Complex correlator ($N = 32$)	1	0	1	160	−18.26	0
Complex correlator ($N = 128$)	1	0	1	896	−29.9	0
MT symbol decoding (SF = 32)	1	0	1	128	−12.4	0
BS symbol decoding (SF = 32)	1	0	1	128	−9.5	0
MT RAKE receiver (SF = 4)	9	0	9	80	−12.6	0
BS RAKE receiver (SF = 8)	5	0	5	50	−2.5	0

operations. Indeed, these scaling operations are inserted to code the fixed-point data with the maximal accuracy. The limited scaling capabilities do not provide the opportunity to scale efficiently the data between two arithmetic operations. In this case, the scaling operation execution time depends on the number of bits to shift. The optimisation process reduces dramatically the scaling operation execution time by moving these operations towards the application inputs. When the scaling operations are located at the application inputs, the execution time of the scaling operations T_{SO} is null. Indeed, the *prescaler* register can scale the inputs when they are loaded from memory, with no supplementary cycle. Nevertheless, these scaling operation transfers degrade the computation accuracy.

These different results show the benefits provided by the optimisation of the scaling operation location and by the guard bits. They underline the necessity to take account of the DSP architecture to obtain an optimised fixed-point specification.

5. CONCLUSIONS

Efficient application implementation in embedded systems requires using fixed-point arithmetic. The reduction of the application time-to-market needs to develop high-level tools which automate the floating-to-fixed-point conversion. In this paper, a new methodology for the floating-to-fixed-point conversion has been proposed. This approach minimises the code execution time under an accuracy constraint. Compared to the previous methodologies, the DSP architecture is taken into account to optimise the fixed-point specification. The fixed-point data types and the scaling operation location are optimised to reduce the code execution time. These two optimisation processes are achieved efficiently thanks to the use of an analytical technique to evaluate the computation accuracy. Indeed, this technique reduces dramatically the optimisation time compared to a simulation-based approach.

Different experiments have been conducted to analyse the efficiency of our approach. The results obtained for the data type selection underline the tradeoff between the accuracy and the code execution time which can be obtained with the recent DSPs. Moreover, the ability of our technique to reduce significantly the execution time with SWP instructions compared to a classical implementation has been demonstrated through the WCDMA receiver example. Indeed, this technique reduces the code execution time by a factor between 1.9 and 3.5. The experiments on scaling operations show that their execution time can become important. The use of guard bits or the optimisation of the scaling operation location reduces significantly the code execution time.

REFERENCES

[1] T. Grötker, E. Multhaup, and O. Mauss, "Evaluation of HW/SW tradeoffs using behavioral synthesis," in *Proceeding of 7th International Conference on Signal Processing Applications and Technology (ICSPAT '96)*, pp. 781–785, Boston, Mass, USA, October 1996.

[2] K.-I. Kum, J. Kang, and W. Sung, "AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors," *IEEE Transactions on Circuits and Syst—Part II*, vol. 47, no. 9, pp. 840–848, 2000.

[3] M. Willems, V. Bursgens, and H. Meyr, "FRIDGE: floating-point programming of fixed-point digital signal processors," in *Proceeding of 8th International Conference on Signal Processing Applications and Technology (ICSPAT '97)*, San Diego, Calif, USA, September 1997.

[4] D. Menard, P. Quemerais, and O. Sentieys, "Influence of fixed-point DSP architecture on computation accuracy," in *Proceeding of 11th European Signal Processing Conference (EUSIPCO '02)*, vol. 1, pp. 587–590, Toulouse, France, September 2002.

[5] S. Kim and W. Sung, "A floating-point to fixed-point assembly program translator for the TMS 320C25," *IEEE Transactions on Circuits and Systems—Part II*, vol. 41, no. 11, pp. 730–739, 1994.

[6] H. Keding, M. Willems, M. Coors, and H. Meyr, "FRIDGE: a fixed-point design and simulation environment," in *Proceeding of IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE '98)*, pp. 429–435, Paris, France, February 1998.

[7] M. Willems, V. Bursgens, H. Keding, T. Grötker, and H. Meyr, "System level fixed-point design based on an interpolative approach," in *Proceeding of 34th ACM/IEEE Design Automation Conference (DAC '97)*, pp. 293–298, Anaheim, Calif, USA, June 1997.

[8] K.-I. Kum, J. Kang, and W. Sung, "A floating-point to integer C converter with shift reduction for fixed-point digital signal processors," in *Proceeding of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '99)*, vol. 4, pp. 2163–2166, Phoenix, Ariz, USA, March 1999.

[9] S. Kim and W. Sung, "Fixed-point error analysis and word length optimization of 8×8 IDCT architectures," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 8, no. 8, pp. 935–940, 1998.

[10] S. Kim, K.-I. Kum, and W. Sung, "Fixed-point optimization utility for C and C++ based digital signal processing programs," *IEEE Transactions on Circuits and Systems—Part II*, vol. 45, no. 11, pp. 1455–1464, 1998.

[11] L. De Coster, M. Ade, R. Lauwereins, and J. Peperstraete, "Code generation for compiled bit-true simulation of DSP applications," in *Proceeding of 11th IEEE International Symposium on System Synthesis (ISSS '98)*, pp. 9–14, Hsinchu, Taiwan, December 1998.

[12] H. Keding, M. Coors, O. Lüthje, and H. Meyr, "Fast bit-true simulation," in *Proceeding of 38th ACM/IEEE Design Automation Conference (DAC '01)*, pp. 708–713, Las Vegas, Nev, USA, June 2001.

[13] H. Keding, F. Hurtgen, M. Willems, and M. Coors, "Transformation of floating-point into fixed-point algorithms by interpolation applying a statistical approach," in *Proceeding of 9th International Conference on Signal Processing Applications and Technology (ICSPAT '98)*, Toronto, Ontario, Canada, September 1998.

[14] D. Menard and O. Sentieys, "Automatic evaluation of the accuracy of fixed-point algorithms," in *Proceeding of IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE '02)*, pp. 529–535, Paris, France, March 2002.

[15] R. Wilson, "SUIF: an infrastructure for research on parallelizing and optimizing compilers," Tech. Rep. CA 94305-4055, Computer Systems Laboratory, Stanford University, Stanford, Calif, USA, May 1994.

[16] F. Charot, F. Djieya, and C. Wagner, "Retargetable compilation in the service of interactive ASIP design," Tech. Rep. 1173, Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), Rennes, France, November 2000.

[17] F. Charot and V. Messe, "A flexible code generation framework for the design of application specific programmable processors," in *Proceeding of 7th IEEE International Workshop on Hardware/Software Codesign (CODES '99)*, pp. 27–31, Rome, Italy, May 1999.

[18] V. Madisetti, *VLSI Digital Signal Processors: An Introduction to Rapid Prototyping and Design Synthesis*, IEEE Press/Butterworth-Heinemann, Boston, Mass, USA, 1995.

[19] D. Menard, R. Rocher, P. Scalart, and O. Sentieys, "SQNR determination in non-linear and non-recursive fixed-point systems," in *Proceeding of 12th European Signal Processing Conference (EUSIPCO '04)*, pp. 1349–1352, Vienna, Austria, September 2004.

[20] G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "Truncation noise in fixed-point SFGs," *IEE Electronics Letters*, vol. 35, no. 23, pp. 2012–2014, 1999.

[21] R. Kearfott, "Interval computations: introduction, uses, and resources," *Euromath Bulletin*, vol. 2, no. 1, pp. 95–112, 1996.

[22] T. W. Parks and C. S. Burrus, *Digital Filter Design*, John Wiley & Sons, New York, NY, USA, 1987.

[23] Texas Instruments Incorporated, *TMS320C54x DSP Reference Set, Volume 1: CPU And Peripherals*, Texas Instruments, Dallas, Tex, USA, January 1999.

[24] Lucent Technologies, *DSP16xx*, Lucent Technologies, Murray Hill, NJ, USA.

[25] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee, *DSP Processor Fundamentals: Architectures and Features*, Berkeley Design Technology, Fremont, Calif, USA, 1996.

[26] B. Ovadia and G. Wertheizer, "PalmDSPCore—Dual MAC and parallel modular architecture," in *Proceeding of 10th International Conference on Signal Processing Applications and Technology (ICSPAT '99)*, Miller Freeman, Orlando, Fla, USA, November 1999.

[27] D. Efstathiou, L. Fridman, and Z. Zvonar, "Recent developments in enabling technologies for software defined radio," *IEEE Communications Magazine*, vol. 37, no. 8, pp. 112–117, 1999.

[28] Analog Device Incorporation, *TigerSHARC Hardware Specification*, Analog Device, December 1999.

[29] Texas Instruments Incorporated, *TMS320C64x Technical Overview*, Texas Instruments, Dallas, Tex, USA, February 2000.

[30] 3DSP, *SP-5 Fixed-point Signal Processor Core*, 3DSP Corporation, Irvine, Calif, USA, July 1999.

[31] CEVA Incorporation, *CEVA-X1620 Datasheet*, CEVA, San Jose, Calif, USA, 2004.

[32] S. Wichman and N. Goel, *The Second Generation ZSP DSP*, LSI Logic Corporation, Milpitas, Calif, USA, 2002.

[33] N. Ghazal, R. Newton, and J. Rabaey, "Predicting performance potential of modern DSPs," in *Proceeding of 37th ACM/IEEE Design Automation Conference (DAC '00)*, pp. 332–335, Los Angeles, Calif, USA, June 2000.

[34] A. Pegatoquet, E. Gresset, M. Auguin, and L. Bianco, "Rapid development of optimized DSP code from a high level description through software estimations," in *Proceeding of 36th ACM/IEEE Design Automation Conference (DAC '99)*, pp. 823–826, New Orleans, La, USA, June 1999.

[35] R. Fletcher, *Practical Methods of Optimization*, John Wiley & Sons, New York, NY, USA, 2nd edition, 1987.

[36] Texas Instruments Incorporated, *TMS320C5x User's Guide*, Texas Instruments, Dallas, Tex, USA, June 1998.

[37] V. Zivojnovic, J. M. Velarde, C. Schläger, and H. Meyr, "DSPStone: A DSP-oriented benchmarking methodology," in *Proceeding of 5th International Conference on Signal Processing Applications and Technology (ICSPAT '94)*, pp. 715–720, Miller Freeman, Dallas, Tex, USA, October 1994.

[38] VLSI Technology, *VVF 3500 DSP Core Rev. 1.2*, VLSI Technology, June 1998.

[39] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proceeding of 32nd ACM/IEEE Design Automation Conference (DAC '95)*, pp. 456–461, San Francisco, Calif, USA, June 1995.

[40] T. Ojanperä and R. Prasad, Eds., *WCDMA: Towards IP Mobility and Mobile Internet*, Universal Personal Communications Series, Artech House, Norwood, Mass, USA, 2002.

Daniel Menard received the Engineering degree and the M.S. degree in electronics and signal processing engineering from the University of Nantes Polytechnic School in 1996, and the Ph.D. degree in signal processing and telecommunications from the University of Rennes, in 2002. From 1996 to 2000, he was a research engineer at the University of Rennes. He is currently an Associate Professor of electrical engineering at the University of Rennes (ENSSAT) and a member of the R2D2 (Reconfigurable Retargetable Digital Devices) Research Team at the IRISA Laboratory. His research interests include implementation of signal processing and mobile communication applications in embedded systems and floating-to-fixed-point conversion.

Daniel Chillet received the Engineering degree and the M.S. degree in electronics and signal processing engineering from ENSSAT, University of Rennes, respectively, in 1992 and in 1994, and the Ph.D. degree in signal processing and telecommunications from the University of Rennes, in 1997. He is currently an Associate Professor of electrical engineering at the University of Rennes (ENSSAT) and a member of the R2D2 (Reconfigurable Retargetable Digital Devices) Research Team at the IRISA Laboratory. His research interests include memory hierarchy, reconfigurable resources, real-time systems, and middleware. All these topics are studied in the context of SoC design for embedded systems.

Olivier Sentieys received the Engineering degree and the M.S. degree in electronics and signal processing engineering from ENSSAT, University of Rennes, in 1990, the Ph.D. degree in signal processing and telecommunications from the University of Rennes, in 1993, and the "Habilitation Diriger des Recherches" degree in 1999. He is currently a Professor of Electrical Engineering at the University of Rennes (ENSSAT). He is the Cohead of the R2D2 (Reconfigurable Retargetable Digital Devices) Research Team at the IRISA Laboratory and is a cofounder of Aphycare Technologies, a company developing smart sensors for biomedical applications. His research interests include VLSI integrated systems for mobile communications, finite arithmetic effects, low-power and reconfigurable architectures, and multiple-valued logic circuits. He is the author or coauthor of over 70 journal publications or published conference papers and holds 4 patents.