Scalable Analytical Accuracy Models for Fixed-Point Arithmetic Implementations

Gaël Deest

Master recherche en informatique

Internship Report

Scalable Analytical Accuracy Models for Fixed-Point Arithmetic Implementations

Auteur
Gaël Deest
Université de Rennes 1

Superviseur:
Steven Derrien
Équipe Cairn

Signal processing algorithms for embedded systems are usually designed and specified with floating-point datatypes, but many embedded architectures do not even feature floating-point operators. Before being implemented in hardware, the algorithm must hence undergo a conversion to fixed-point arithmetic. This conversion is a tedious process and can cause a degradation in the accuracy of the algorithm. This degradation must be evaluated in order to guarantee the proper functioning of the system. Also, a typical requirement is to minimize the cost of the system as long as the accuracy criterion is satisfied. It is called *wordlength optimization*.

Until recently, the only methods available for accuracy evaluation were based on bit-accurate simulations. Simulation times prohibited the use of automatic optimization algorithms. During the last decade, new analytical methods have emerged. These methods still face severe scaling limitations and cannot handle parametric programs.

In this work we study the relevance of the polyhedral model and of polyhedral intermediate representations for the development of a new analytical technique for accuracy estimation. We show that the formalism of *Systems of Affine Recurrence Equations* (SAREs) can be used to perform noise analysis. We derive a method and a set of program transformations that allow us to handle some classes of programs, and even some recursive filters.

Keywords: signal processing, embedded systems, fixed-point arithmetic, quantization noise, wordlength optimization, static analysis, polyhedral model

Contents

1 Introduction **4**

2 Fixed-Point Arithmetic for Embedded Systems **7**
2.1 Design of Embedded Systems . . . 7
2.1.1 Design Constraints of Embedded Systems . . . 7
2.1.2 Representation of Real Numbers . . . 8
2.1.3 Design Flow and Implementation of Signal Processing Systems . . . 9
2.2 Float-to-Fix and Wordlength Optimization . . . 11
2.2.1 Fixed-Point Errors . . . 11
2.2.2 Wordlength Optimization . . . 12
2.3 Techniques for Accuracy Analysis . . . 12
2.3.1 Simulation-Based Techniques . . . 13
2.3.2 Analytical Techniques . . . 13

3 Problem Statement **15**
3.1 Limitations of Current Techniques . . . 15
3.1.1 Scaling Issues . . . 16
3.1.2 Inability to Handle Parametric Programs . . . 17
3.1.3 Noise Power Formula Derivation . . . 17
3.2 Goal of Our Work . . . 18

4 Proposed Approach **19**
4.1 Background . . . 19
4.1.1 Polyhedral Model and SCoPs . . . 19
4.1.2 From SCoPs to PRDGs . . . 20
4.1.3 SAREs and the Alpha Language . . . 21
4.2 Overview of Our Approach . . . 22
4.3 SARE Transformations . . . 23
4.3.1 Dealing With Aging Buffers . . . 23
4.3.2 Accumulations . . . 27
4.3.3 Recursivity . . . 28
4.4 Noise Model Computation . . . 28
4.5 Non-Recurrent Noise Models for Recursive Algorithms . . . 30
4.6 Safety . . . 31
4.7 Implementation Status . . . 32

5 Conclusion and Future Work **33**

1 Introduction

Embedded systems such as smartphones or MP3 players must satisfy a set of seemingly incompatible constraints: size, cost, performance, accuracy, energy use, etc. The challenge faced by the designers is to reconcile these requirements. It is a time-consuming process, but time is a scarce resource. Manufacturers are under pressure to release their products as fast as possible and a small delay can result in significant losses of market share. The growing importance of *Time-To-Market* (the delay between the conception of a product and its release) has generated an interest for new tools that could reduce the design time and has driven the research on *High-Level Synthesis*: the automatic generation of a hardware description from a high-level specification.

Many embedded systems implement signal processing algorithms (*e.g.*, video decoding or wireless communication protocols). These algorithms are computationally-intensive and operate on real numbers, which must be somehow represented in hardware. The choice of this representation is one of the many factors that can have an impact on the performance of a signal-processing system. The designer can choose between fixed-point and floating-point arithmetics. For a given wordlength, floating-point architectures can handle a wider range of numbers and offer good accuracy. Numerically stable algorithms can hence be implemented without concern. Unfortunately, the latency, area cost and power consumption of floating-point operators prohibit their use in most highly constrained embedded platforms and fixed-point arithmetic is preferred in this context.

The use of fixed-point arithmetic comes with numerical errors: numbers must be scaled to prevent overflows or to limit the wordlength of data samples and intermediate results. Numbers are rounded to a lower unit of precision; this operation is called a *quantization*. Quantizations can occur at different stages in the computation flow, with varying impact on accuracy. To quantify this impact, quantization errors are modeled as additive *noises* (*i.e.*, random signals), called *quantization noises*. These noises are random variables, propagated and possibly amplified by subsequent operations.

A typical embedded system designer tries to minimize the overall design cost (chip area or energy budget) enforcing some accuracy criterion (*e.g.*, maximum noise power). This problem is called *wordlength optimization* (WLO). It is of NP-hard complexity [7], but efficient heuristics can be applied [15]. During the optimization process, the accuracy of the design must re-assessed after every new fixed-point encoding decision. Unfortunately, conventional accuracy evaluation techniques use bit-level accurate simulations of the hardware design. They are hence too slow to be used in a fully automated process, as a new set of simulations would have to be run for each optimization iteration. For this reason, WLO is almost always done manually by experienced designers. It is a long and tedious task which can take up to 30% of the total development time [18]. In addition, because it is performed in an ad-hoc way, there is a risk of missing some trade-off opportunities that could be discovered by more systematic techniques.

During the last decade, promising analytical methods have emerged [13] [6] that could open the path to fully automatic WLO. The fundamental idea is to derive a *closed-form expression* of quantization noise power as a function of the chosen fixed-point formats. A closed-form expression is an expression that can be computed in constant time. This expression, computed once and for all, can then be used to perform WLO without resorting to simulations.

These methods operate on a *Signal Flow Graph* (SFG) representation, which is a classical model used in signal processing. Current analytical tools either operate from an explicit representation of the system in the form of a SFG [6] or try to infer an SFG model from an algorithmic specification in C or MATLAB, such as ID.FIX[1]. The ability to operate directly over an algorithmic specification is considered as being a very important point, yet current WLO tools suffer from severe limitations. In particular, they require the flattening of the program by unrolling the loops, which makes them very sensitive to the number of iterations and they face severe scaling issues. Moreover, they are not applicable if some program parameters, like iteration bounds, are unknown at design time. Therefore, a lot of open problems remain to be addressed before such approaches can be fully integrated within a compiler.

The limitations of current techniques can be explained by the low level of abstraction on which they operate. Indeed, during the flattening of the program, all the information about the control flow and the regularity of the program is lost. However, research in optimizing compilers has shown that high-level language constructs can be used to reason about programs much more efficiently than on assembly code. In particular, loop nests have been studied for decades [9]. Powerful loop transformations and parallelization techniques have been proposed (*e.g.* [1]), which don't require loop flattening and can even be applied when iteration bounds are unknown at compile-time. It is plausible that a similar rise in abstraction and the use of more appropriate intermediate representations will enable the emergence of more powerful and scalable accuracy evaluation techniques. To the best of our knowledge, though, very few authors [4] have followed this path and tried to apply the formalisms developed by the compiler community to this problem.

The *polyhedral model* is such a formalism. It can be used to represent, analyze and transform a small but important class of programs known as *Static Control Parts* (SCoPs). Many signal processing algorithms fall into this class. This model provides several concise and expressive intermediate representations that may be used in place of SFGs.

The main goal of our work was to study whether the expressivity of the polyhedral model could be leveraged to tackle the aforementioned limitations of analytical methods. Here is a summary of our results: for SCoPs, noise power can be expressed very naturally as a system of recurrence equations. We proposed and implemented a set of program transformations that allow us to compute a noise model of the system. For most non-recursive, uni-dimensional filters, this model can be further simplified to a simple closed-form expression. For recursive filters, the model often takes the form of a linear recurrence that can be solved with the help of a symbolic solver, and we are once again able to give a non-recurrent closed-form solution.

The remaining of this report is organized as follows. In chapter 2, we present the

[1] http://idfix.gforge.inria.fr

specific context of embedded systems and the difficulties faced by designers regarding fixed-point arithmetic. We present the techniques currently available to overcome them, with emphasis put on analytical methods. We give a few essential definitions along the way. In chapter 3, we expose through an example why current analytical techniques are not completely satisfactory. In chapter 4, we present our results. We show that the use of an equational formalism from the polyhedral model and a set of high-level transformations allow us to give, for some classes of programs, an expression of noise power as a function, for example, of the number of iterations. In chapter 5, we summarize our work, put it in perspective and sketch a few ideas for future work.

2 Fixed-Point Arithmetic for Embedded Systems

The goal of this chapter is to familiarize the reader with the characteristics of embedded systems and one of the many challenges faced by designers: the use of fixed-point arithmetic.

In section 2.1, we will briefly review the constraints of embedded systems, expose the differences between fixed-point and floating-point numbers, and explain why fixed-point arithmetic if often preferred in this context. In section 2.2, we will expose the problem of *floating-point to fixed-point conversion*, and the techniques currently available to address it.

2.1 Design of Embedded Systems

Designing an embedded system implies reconciling many incompatible requirements. The representation of real numbers can have a major impact on the cost, size, performance, energy use and accuracy of these systems. In this section, we will see why embedded systems are usually implemented with fixed-point arithmetic, in spite of the numerical merits of floating-point arithmetic.

2.1.1 Design Constraints of Embedded Systems

Embedded systems are subject to numerous design constraints. Some of them are inherent to their function or embedded nature. Others are the consequence of external requirements such as commercial profitability. The most important constraints are:

Cost The design must not be too expensive to be sold at a competitive price.

Size Most portable equipments, such as smartphones, are relatively small.

Energy Consumption If the design requires too much energy, the autonomy of the system will be affected. Also, the heat produced by power dissipation might exceed reasonable limits.

Performance Many multimedia and communication devices must be able to encode or decode high resolution data streams on-the-fly.

Real-Time Constraints Some devices are subject to more or less strict timing constraints. For example, a video frame must be decoded within a certain delay to prevent visual glitches.

Accuracy While some degradation of the quality of the service may be acceptable (*e.g.*, for voice decoding), accuracy must not drop below certain limits.

Time-To-Market Last but not least, consumer electronics products must reach the market quickly. Outmoded devices cannot be sold at a profitable price.

Many of these constraints are in conflict. For example, the need for performance requires the use of high-end components that are expensive, use more silicon area and need more energy. Usually, subtle trade-offs must be made to satisfy all the requirements. One of the many factors that can affect the quality of embedded systems is the representation of real numbers.

2.1.2 Representation of Real Numbers

Many embedded devices implement signal processing algorithms and perform computationally intensive tasks. The representation of real numbers can have a significative impact on their cost, size, performance, energy consumption and accuracy. Two options are available: floating-point and fixed-point arithmetics. In the folowing, we recall the definition of these representations and explain why fixed-point arithmetic is usually preferred.

Floating-Point Numbers

Broadly speaking, floating-point numbers are the binary equivalent of the scientific notation. The general schema of a floating-point format is shown in Figure 2.1.

Sign (1 bit)	**E**xponent (e bits)	**M**antissa (m bits)

Figure 2.1: Anatomy of a floating-point format (as in the norm IEEE-754). The value of a number is given by the formula: $(-1)^{\mathbf{S}} \times \mathbf{M} \times 2^{\mathbf{E}-\text{shift}}$, where shift $= 2^{e-1} - 1$ (the exponent is a signed integer, but is stored as an unsigned integer by shifting all the values by a positive constant).

The strength of floating-point numbers lies in the fact that the exponent can be adjusted to represent either small numbers with high precision or large numbers with less accuracy. In other words, floating-point numbers can represent a wide range of values, but the available bits are never wasted: when the number is small, most of the bits of the mantissa are used to encode digits after the point. For this reason, the relative encoding error is small and bounded. They are thus suited for the implementation of numerically stable algorithms.

The drawbacks of these good numerical properties are the size, cost, latency and power consumption of floating-point operators.

Fixed-Point Numbers

The fixed-point representation is probably the simplest way to represent real numbers in a processor. Simply put, fixed-point numbers are nothing but integers scaled with an implicit power of two, called the scaling factor:

$$2^k \times \boxed{\text{integer value}}$$

Most often, k is negative, and k bits are reserved for the *fractional part* of the number (digits after the point). The other bits are used to encode the *integral part*. However, it needs not be the case.

Fixed-point numbers lack the versatility of floating-point numbers: the scaling factor cannot be adjusted dynamically to mimick the effect of the variable exponent. Also, they are not as convenient: it is necessary to deal with the scaling factor explicitly in the design. Indeed, when the product of two fixed-point numbers is computed, the scaling factor of the result is the product of the scaling factors of the operands. Hence, the scaling factor may not be defined once and for all and considered constant over the computation.

However, since fixed-point numbers are nothing but integers in a disguise, arithmetic operations are fast. Integer operators are also much smaller than floating-point operators and require less energy. These desirable properties explain why they are often favoured by embedded systems designers.

2.1.3 Design Flow and Implementation of Signal Processing Systems

Many embedded systems implement signal procesing algorithms. Algorithmic exploration is done by domain experts (image processing or telecommunication engineers). These experts do use high-level tools or languages to perform algorithmic prototyping. Such tools (MATLAB) operate by default on floating-point data. Once the algorithm is defined, it has to be implemented on the target hardware, where floating-point may not be available. The algorithm then needs to undergo the so-called *floating-point to fixed-point conversion step* (float-to-fix).

Float-to-fix is usually performed by assigning a fixed-point format (wordlength, scaling factor) to each variable in the implementation. It is a tedious task, as no support for fixed-point data is usually available in programming languages. The designer must also ensure that the functionality of the system is preserved. Indeed, fixed-point arithmetic is inherently less precise than floating-point arithmetic and can introduce significant encoding errors. These errors can have a strong impact on the quality of the design and change the semantics of the original algorithm. This impact is quantified by the mean of an appropriate metric and compared to some pre-defined accuracy constraint. A designer will typically try to minimize the cost of the design as long as this constraint is satisfied. This process is known as *wordlength optimization* (WLO). The whole float-to-fix conversion process is represenced in Figure 2.2.

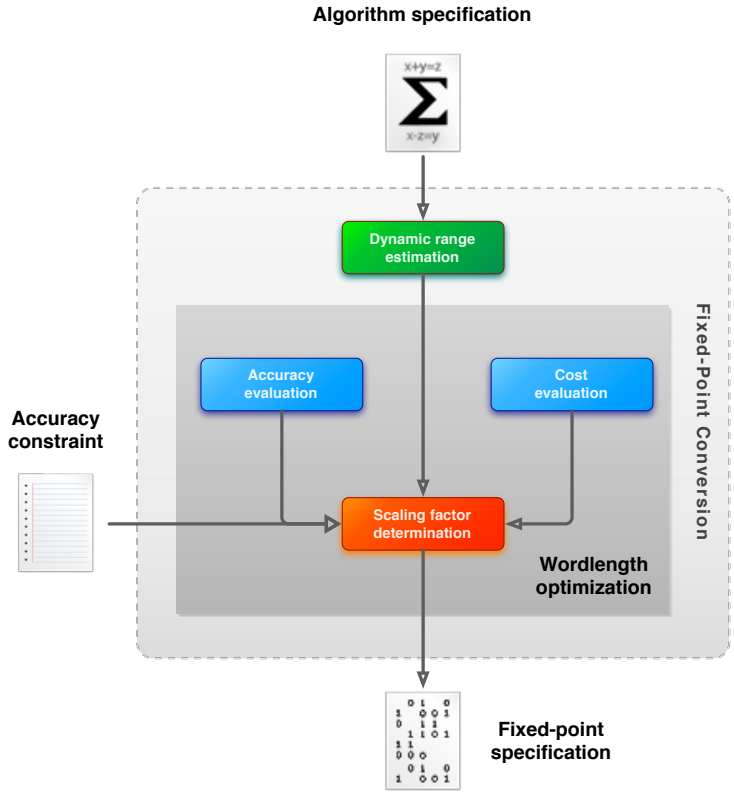


Figure 2.2: Floating-point to fixed-point conversion process. The range of values reached by each variable in the algorithm is first evaluated. Wordlength optimization is then performed under an application-specific accuracy constraint.

2.2 Float-to-Fix and Wordlength Optimization

In this section, we review the different types of errors that can arise from the use of fixed-point arithmetic. We then expose the principles of *wordlength optimization* (WLO).

2.2.1 Fixed-Point Errors

The use of fixed-point arithmetic can result in two kind of errors.

Overflow Errors

Overflow errors occur when the result of a computation exceeds the range of values supported by the register or the memory cell in which it must be stored.

They can also occur in floating-point arithmetic. However, fixed-point formats can usually represent a narrower range of values than floating-point formats. It follows that fixed-point architectures are much more vulnerable to this kind of errors.

Quantization Errors

A *quantization* is the fact of constraining a signal to a lower resolution. For a signal-processing algorithm, we distinguish two kind of quantizations:

Quantization of Inputs In the floating-point specification, the precision of input samples can be considered infinite. In the fixed-point implementation, all the bits below a certain significance are dropped. This is a first source of errors, called *quantization of inputs*.

Internal Quantizations Consider an algorithmic specification containing the following instruction: `a = b * c`. Suppose that for all three variables, a scaling factor of 2^{-4} is chosen. The result of the product has a scaling factor of 2^{-8}: this means that the four least significant bits of the result will be dropped when it is stored in variable `a`. Such a rescaling is called an *internal quantization*.

Quantization errors are naturally propagated by arithmetic operations. The main rules are summarized in Table 2.1.

Table 2.1: Error propagation rules

Operation	Propagated noise	Comments
$x + y$	$e_x + e_y$	
$x + y$	$e_x - e_y$	
$x \times y$	$xe_y + ye_x + e_xe_y$	The term e_xe_y is usually considered neglectable.

Error Metrics

It is customary to model quantization errors as a random additive noise affecting the output of an operation:

$$\widehat{\text{output}} = \text{output} + \text{error}.$$

Two metrics are commonly used to quantify this noise:

Noise Power is simply defined as its variance: $P_{\text{noise}} = \text{Var}(\text{noise})$.

Signal to Quantization Noise Ratio is defined as the ratio : $\text{SQNR} = \frac{P_{\text{output}}}{P_{\text{noise}}}$.

2.2.2 Wordlength Optimization

For *application-specific integrated circuits* (ASICs), custom processors designed for a specific task, wordlength can be chosen arbitrarily. In this context, wordlength optimization is the process of minimizing a given *cost function* by associating a fixed-point format (wordlength, scaling factor) to each variable in the program, respecting a given *accuracy constraint*.

For *digital signal processors* (DSPs), semi-generic processors optimized for integer computations, wordlength is imposed by the target architecture (*e.g.*, 8, 16 or 32 bits), and the only variable is the scaling factor. One may try to choose the smallest DSP processor on which the accuracy constraint can be respected, or to maximize the accuracy for a given processor.

During WLO, two important aspects must be considered. First, we must guarantee the absence of overflows, as they can have a dramatic impact on the computation. Secondly, we must be able to assess the impact of different fixed-point encoding decisions on the accuracy of the program output.

Prevention of Overflows

In order to prevent overflows, the most common approach is to determine the range of values of each variable in the program, called its *dynamic range*. Is is then possible to ensure at design-time that enough bits are available to prevent overflows.

The dynamic range can be determined using static techniques such as *interval analysis* [14]. Another approach, based on the polyhedral model and the $(\max, +)$ algebra, was proposed by Cachera et al. [3]. If some transient overflow errors can be tolerated, simulation usually provides tighter bounds than static analysis.

Another way to prevent overflows, or rather to limit their impact on the computation, is to use *saturated arithmetic*. In saturated arithmetic, when the result of an operation exceeds the maximum (*resp.* minimum) possible value, it is simply replaced by this maximum (*resp.* minimum).

2.3 Techniques for Accuracy Analysis

Two main families of techniques can be distinguished: simulation-based techniques and analytical methods.

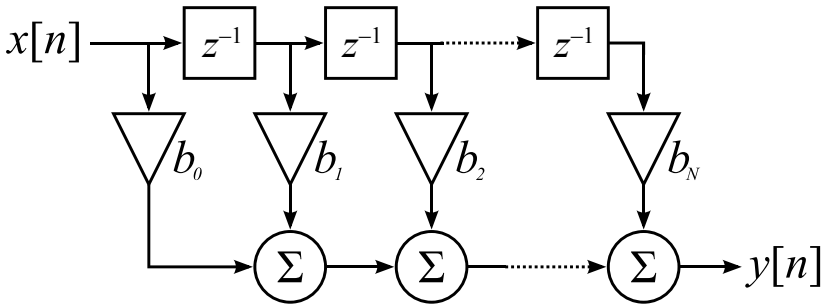


Figure 2.3: SFG for *Finite Impulse Response* filter. This filter computes the function: $y[n] = \sum_{i=0}^{N} b_i \times x[n-i]$. Σ nodes represent sums. Triangle nodes represent a product by a constant. z^{-1} nodes represent *delays*: at iteration n, the output of a delay node is its input at iteration $n-1$.

2.3.1 Simulation-Based Techniques

Simulation techniques are probably the most used. They are based on bit-level accurate simulations of the fixed-point system. A large number of simulations is performed, and the results are compared with those of the reference floating-point implementation. The statistical moments of the noise (mean, variance) are then computed.

The main drawback of these techniques is their poor performance. Simulation of fixed-point mechanics implies a significant computational overhead.

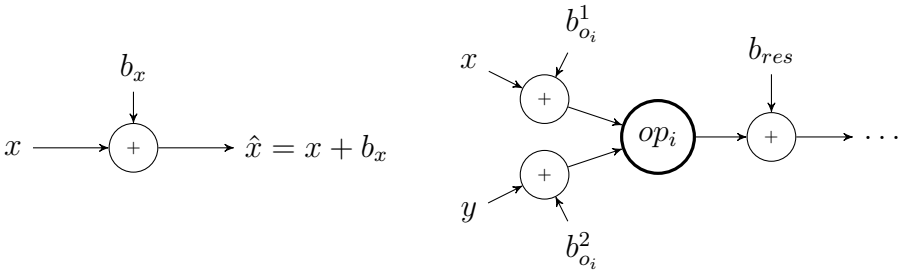
2.3.2 Analytical Techniques

The idea of these techniques is to derive an expression of noise power analytically. Current techniques rely on a *Signal Flow Graph* (SFG) representation. SFGs are a classical signal processing formalism. An example of SFG is shown in Figure 2.3.

Some techniques take directly a SFG as input [6] while others try to infer a SFG from a C program [11]. Inferring a SFG from a C program implies flattening its control flow: all the loops must be unrolled.

Early techniques [12] focused on recursive and non-recursive *Linear Time Invariant* (LTI) systems, by extracting a *transfer function* from a SFG noise model. The construction of this noise model from the original SFG is illustrated in Figure 2.4. Also, a noise power expression could be computed for non-recursive non-LTI filters, using simple noise propagation rules.

Other techniques based on perturbation theory rely on the fact that quantization noise is relatively small to linearize the noise propagation function of operators [6]. Finally, recent techniques are able to handle all recursive and non-recursive filter expressed with smooth operators [16].



(a) Input noise model (b) Operation noise model

Figure 2.4: Noise model construction for a SFG.

3 Problem Statement

The goal of this chapter is to expose the problem we are trying to address. In section 3.1, we study the limitations of current analytical techniques. We show that many regular programs cannot be analyzed, even though a noise power equation is easy to derive manually. In section 3.2, we present the goal we set ourselves for this work.

3.1 Limitations of Current Techniques

We base our discussion on a *stencil computation*. Stencil computations form a very general class of algorithms and are ubiquitous in image or video processing (optical flow estimation, feature point extraction, etc). In a stencil computation, the cells of an array are repeatedly updated according to a fixed geometrical pattern. Each new value is computed using the values of the neighbor cells.

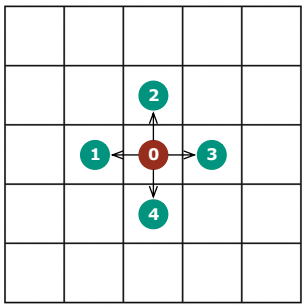


Figure 3.1: Computation pattern of the stencil. Each pixel is replaced by a weighted sum of the values of 5 points: itself and its top, bottom, left and right neighbors. The digit in the circle indicates the index of the coefficient.

Our example is an iterated image filter. At each iteration, each pixel is replaced by a weighted sum of the values of its neighbors. The pattern is represented in Figure 3.1. We use cyclic boundary conditions: pixel coordinates are interpreted modulo the dimensions of the image. For example, in a $W \times H$ image, pixel $(-1, -2)$ is equivalent to pixel $(W - 1, H - 2)$. Cyclic filters are very common for the analysis of repetitive structures and their regularity makes them easy to understand. As such, they constitute a good example.

The C code of the filter is provided in Figure 3.2. One can note than the number of iterations and the dimensions of the image are left blank. By playing on these parameters, we will progressively increase the complexity of our program, pinpointing the limitations of current techniques as they appear. Ultimately, the problem will become impossible to solve with these techniques. In subsection 3.1.3, we nonetheless show that an analytical expression of noise power is easy to derive.

```
// Convenient macro to get a positive modulo.
#define MOD(a,b) (a%b+b)%b

float c[5] = { ... };
int W, H, N;
float img[W][H], tmp[W][H];

for (int n=0; n<N; n++) {
  // For each pixel in the image, compute the
  // weighted average and store in tmp.
  for (int i=0; i<W; i++)
    for (int j=0; j<H; j++) {
      tmp[i][j] = c[0] * img[i][j] +
        c[1] * img[MOD(i-1,W),j] + c[2] * img[i,MOD(j-1,H
)] +
        c[3] * img[MOD(i+1,W),j] + c[4] * img[i,MOD(j+1,H
)] ;
    }

  // Copy tmp over img for the next iteration.
  for (int i=0; i<W; i++)
    for (int j=0; j<H; j++)
      img[i][j] = tmp[i][j];
}
```

Figure 3.2: Code for the weighted sum stencil.

3.1.1 Scaling Issues

If the dimensions of the image and the number of iterations are compile-time constants, they can be propagated and the loops can be unrolled. It is then possible to build a SFG from the program. The size of this SFG obviously depends on the values of N, W and H. Let us compute the number of its nodes as a function of these parameters.

At each iteration, the new value of each pixel is computed with 4 sums and 5 products. There are $W \times H$ values to compute per iteration. The SFG also contains $W \times H$ nodes for the pixels of the original image, plus 5 for the coefficients. Finally, the number of

Table 3.1: Number of nodes of the SFG for the weighted sum stencil

N	W	H	$f(N, W, H)$
4	16	16	$9,477$
10	64	64	$372,741$
50	320	240	$34,636,805$
100	1920	1080	$1,868,313,605$

nodes is given by:

$$f(N, W, H) = 5 + WH(1 + 9N).$$

In Table 3.1, we computed the number of nodes for a few sets of realistic parameters. Note that the largest dimensions in this table correspond to a standard, $1080p$ high-definition image. For 100 iterations, the SFG contains almost 2 billion nodes. At 1 byte per node, 1.74 GiB would be required to store the SFG, without counting edges.

For this reason, existing analytical methods face major scaling issues. It turns out, from experimentation with ID.Fix, that computing the closed-form for a 512 points *Fast Fourier Transform* requires a whole day. The 1024 points FFT is out of reach.

3.1.2 Inability to Handle Parametric Programs

In the previous subsection, we assumed that some program parameters, like the number of iterations, were compile-time constants. Unfortunately, it is not always the case and existing techniques then not be applied.

In fact, it is sometimes possible to derive a noise power expression as a function of the parameters, as we will show in the next subsection. As we will see, some parameters do not even affect noise power.

3.1.3 Noise Power Formula Derivation

In the following, we will show that it is straightforward to derive a noise power expression for the program in Figure 3.2.

Recall that the power of a signal x is simply its variance. We will need these two results from elementary probability theory:

1. If two random variables X and Y are uncorrelated, $\mathrm{Var}(X+Y) = \mathrm{Var}(X)+\mathrm{Var}(Y)$.
2. If a is a constant X a random variable, $\mathrm{Var}(aX) = a^2\mathrm{Var}(X)$.

It is customary to assume that quantization noises are uncorrelated. Recall Table 2.1 that the error propagated by a sum is simply the sum of the errors. It follows from result 1. that the same holds for noise power.

Let us derive a noise power formula for our example. For ease exposition, we will only consider input quantization noise, assuming that no internal quantizations occur. It is obviously a naive hypothesis, but it does not change the discussion in a fundamental way.

Our goal is to define a function $P(N, W, H, i, j)$ providing noise power as a function of the parameters and the coordinates of the pixels. It is reasonable to assume that noise power is the same for every pixel in the original image; we will denote this input noise σ^2.

We obviously have:

$$P(0, W, H, i, j) = \sigma^2.$$

Now, let $N > 0$. Under our hypotheses:

$$P(N, W, H, i, j) = c_0^2 P(N-1, W, H, i, j) + c_1^2 P(N-1, W, H, (i-1) \bmod W, j) + \dots$$

Since input noise power is the same for every pixel, we can prove inductively that the value of the formula depends only on N. Omitting W, H, i and j we can write:

$$\begin{aligned} P(0) &= \sigma^2 \\ P(N) &= \left(\sum_{i=0}^{4} c_i^2 \right) P(N-1) \qquad\qquad (N > 0) \end{aligned}$$

This is a trivial recurrence, whose closed-form solution is simply:

$$P(N) = \left(\sum_{i=0}^{4} c_i^2 \right)^N \sigma^2.$$

3.2 Goal of Our Work

In this chapter, we proved that current techniques were unable to handle programs for which a noise power expression is nonetheless easy to derive if the structure of the program is conserved. We explicitly formulated noise power at iteration N as a function of noise power at iteration $N-1$. The SFG representation does not allow such reasoning because the control flow information is lost during loop unrolling. Also, loop unrolling is not possible if the number of iterations is unknown, but reasoning about the control flow allowed to express noise power as a function of these unknown parameters.

Of course, the noise expression in subsection 3.1.3 was derived in a very ad-hoc way. At first glance, it is not clear whether it can be easily generalized to a wide class of programs. The goal of this work was to study the use of alternative program representations to develop a new analytical noise estimation technique. Our results are presented in the next chapter.

4 Proposed Approach

In chapter 3, we exposed some limitations of current analytical methods for accuracy evaluation. These limitations are a direct consequence of the Signal-Flow Graph representation, which requires that the algorithm specification be fully flattened before computation. During this flattening, a lot of information is lost about the structure of the program. As was showed in subsection 3.1.3, this structure can be leveraged to perform noise analysis on programs for which current approaches are either unapplicable or fail to give an answer in a reasonable time.

We formalize this idea within a theoretical framework known as the *polyhedral model*. This model is well known in the optimizing compiler and automatic parallelization communities. However, to the best of our knowledge, only very few works have addressed the problem of accuracy estimation in this context.

In this chapter, we first expose the background knowledge required to understand our solution. We then present our method through a series of examples. We conclude with the status of our implementation.

4.1 Background

The polyhedral model is a program analysis and transformation framework. As it forms the basis of our method, we give here the fundamental notions that will be used throughout this chapter.

4.1.1 Polyhedral Model and SCoPs

The polyhedral model is applicable to a restricted class of program called *static control parts* (SCoPs). A SCoP is a maximal sequence of consecutive statements in which all loop bounds and array accesses are affine functions of the surrounding loop iterators and the parameters of the program (variables whose value does not change within the SCoP).

Example 4.1.1. The following loop nest is a SCoP:

```
for (i=0; i<N; i++)
  for (j=i; j<2*N+1; j++)
    S0: a[i+j] = b[2*i+j] * c[N-i];
```

But the following is not:

```
for (i=0; i<N; i++) {
  N--;
```

```
  for (j=0; j<2*N+1; j++)
    S1: a[i+j] = b[2*i+j] * c[N-d[i]];
}
```

Indeed, in the first example, N is a parameter, but not in the second one: the bound of the outer loop is not an affine function of parameters and loop iterators. Also, note that statement `S1` contains a data-dependent access to array `c`: the control is not *static*.

At first glance, these restrictions can seem very severe, but a large number of programs from a variety of fields (signal processing, linear algebra, numerical analysis, dynamic programming) happen to be SCoPs. It turns out that this class of programs is wide enough to cover many applications while lending itself to automated analysis.

4.1.2 From SCoPs to PRDGs

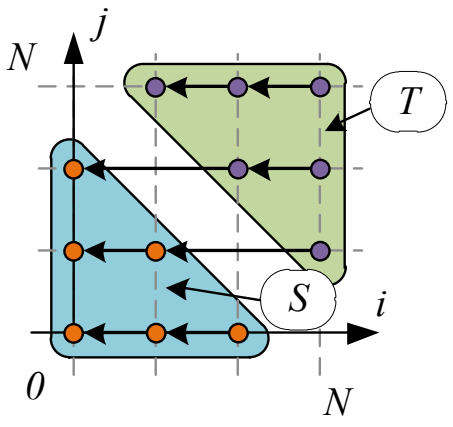
A *polyhedral reduced dependency graph* (PRDG) is a graph capturing all the information about data dependencies in a SCoP. Its sole purpose is to capture dependency information between statement instances; it does not model the semantics of the operations, as these semantics are not needed to perform most loop optimizations.

The goal of this subsection is to give an idea of the construction of a PRDG. The interested reader can refer to Feautrier [8] for further information.

Figure 4.1a contains a SCoP with two statements (`S` and `T`). To each of these statements, we can associate a polyhedral *domain*: the subset of $\mathbf{Z}^2$ whose points correspond to the values taken by i and j when the statement is executed. Each point in a domain represents a single execution of the statement, called an *instance* of that statement or an *operation*. The domains of `S` and `T` are represented in Figure 4.1b.

```
for (i=0; i<N; i++) {
  for (j=0; j<N-i; j++)
    S: t[i+1][j] = f(t[i][j]);
  for (j=N-i+1; j<=N; j++)
    T: t[i][j] = g(t[i-1][j]);
}
```

(a) SCoP



(b) Domains and dependencies ($N = 3$)

Figure 4.1: Program, statement domains and dependencies

Note that `S` and `T` both read and write in the same array `t`. Thanks to the regularity of SCoPs, it is possible, for each statement, to find the producer of the value being read

as a function of i and j. In other words, we can build a function giving, for each instance I of a statement, the last operation writing in the array cell read by I. It is called *exact array dataflow analysis*. In Figure 4.1b, data dependencies between statement instances are represented as arrows, from consumer to producer.

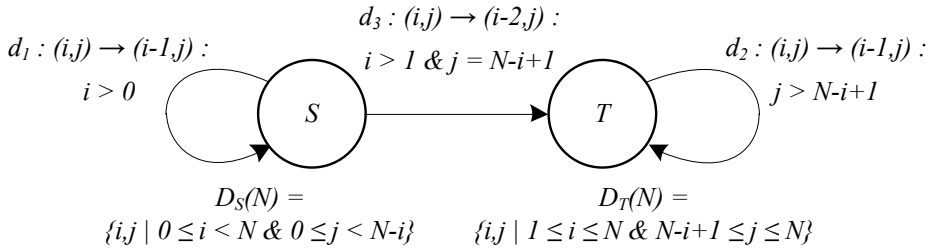


Figure 4.2: PRDG of the SCoP from Figure 4.1a

Exact knowledge of data dependencies allows to build the PRDG of a SCoP. The nodes of a PRDG represent the statements of the SCoP. Edges represent dependencies, and are labeled with *(affine) dependency functions*, defined on a subset of the statement domain. The PRDG of our example is given in Figure 4.2.

Nowadays, PRDGs are used in modern, production-quality compilers as an intermediate representation to perform program transformations such as automatic parallelization. Polyhedral transformation take advantage of mathematical properties of polyhedral domains and affine functions. We use the *Integer Set Library* [19] for manipulating polyhedral objects in our work.

4.1.3 SAREs and the Alpha Language

PRDGs are not sufficient to perform noise analysis. Indeed, quantization noise propagation depends on the arithmetic operation. PRDGs bring information on the dataflow of the program, but not on the operations that are actually performed.

Extending a PRDG with the semantics of the operations is actually equivalent to building a *system of affine recurrence equations* (SARE). A SARE contains all the information of the PRDG, plus semantics.

To represent SAREs, we use the syntax of the *Alpha* language. Alpha was initially developed twenty years ago for the design of systolic arrays [10]. Since then, several flavours of Alpha have been implemented, with minor syntactic differences and some additional features. The version we use is AlphaZ [21].

The core of an Alpha program is a set of equations for multidimensional variables defined over polyhedral domains. Equations can contain *case statements*, to provide different expressions for disjoint subsets of the domain.

Figure 4.3 contains an excerpt of an Alpha program performing a LU *matrix decomposition*. All the reader needs to know to understand this example is that, under certain conditions, a square matrix A may be decomposed as the product of a lower-triangular matrix L and an upper-triangular matrix U, such that $A = LU$. This program takes as input a square matrix of parametric dimension $N \times N$ (N is a parameter of the program) and computes the values of L and U.

```
affine LUD {N|N>0}
given
  float A {i,j|1<=(i,j)<=N};
returns
  float L {i,j|1<i<=N && 1<=j<i};
  float U {i,j|1<=j<=N && 1<=i<=j};
through
  U[i,j] = case
    {|1==i} : A[i,j];
    {|1<i} : A[i,j] - reduce(+, [k], L[i,k]*U[k,j]);
  esac;
  L[i,j] = ... ;
.
```

Figure 4.3: Excerpt of SARE for LU matrix decomposition

Consider the equation for matrix U. It is composed of a case-statement with two branches. The first only applies if the index `i` is equal to 1: it states that every element of the first row is equal to the corresponding element in matrix A. The second branch gives the expression for the other rows. The *reduce* expression corresponds to an accumulation. In mathematical notation, the whole equation reads as:

$$U_{i,j} = \begin{cases} 1 = i \leq j & A_{i,j} \\ 1 < i \leq j & A_{i,j} - \sum_{k=1}^{i-1} L_{i,k} U_{k,j} \end{cases}.$$

Note that in Alpha, expressions are automatically constrained to the domain on which they are defined, which makes the syntax very concise. In particular, the bounds of the summation need not be stated explicitly. Indeed, the lower-triangular domain of L implies that $k < i$. We can deduce that k ranges from 1 to $i - 1$.

Presenting Alpha in detail is not possible here. The interested reader can consult [20].

4.2 Overview of Our Approach

A representation of the flow we propose is presented in Figure 4.4. Our flow uses a C program as input. This program must be a SCoP. If it isn't, an error is reported to the user. Dataflow analysis is then performed and the program is converted to Alpha.

Before a model of the quantization noise behavior of the program can be computed, we first need to apply a set of transformations. Indeed, if the PRDG of the SCoP is acyclic, building this model is an almost trivial task. Unfortunately, it is rarely the case in practice and our PRDG contains several cycles. Some of them can be eliminated: these are the cycles due to *accumulations* and to *aging buffers*, a common idiom in signal processing algorithms in which buffer values are shifted at each iteration as in a shift register. We detect accumulations and replace them by *reductions*, a convenient construct of Alpha.

Cycles due to aging buffers can be eliminated by computing the *transitive closure* of a polyhedral relation. Other kind of cycles will be handled at latter stage, when possible.

Once these cycles have been eliminated, we can build a *noise abstraction* of the program. The idea is to build a SARE describing the behavior of the program at the quantization noise power level. At first glance, our technique may seem reminiscent of *abstract interpretation*. However, we must point out that our technique does *not* always provide safe abstractions, as will be shown in section 4.6.

This noise model must be simplified after computation. Our goal is to reduce the number of equations. This can be achieved by *inlining*, that is, replacing references to other equations by their definitions. For non-recursive filters with a single output stream, it is always possible to reduce the system to a single equation, with possibly several case branches. For recursive filters, inlining is not sufficient because of the cyclic nature of the PRDG. In some cases, however, the noise model takes the form of a linear recurrence that may be solved exactly with symbolic solvers such as MAXIMA[1].

4.3 SARE Transformations

Consider the program in Figure 4.5. It implements an *infinite impulse response* (IIR) filter. An IIR filter takes as input a stream of values and produces another one by computing a weighted sum of the last input and output values.

A simplified PRDG of this program is shown in Figure 4.6. It contains several cycles. We will see that the program can be transformed to eliminate most of them. We can distinguish between cycles that are simply artefacts of the programming language and those, more problematic, which are due to the inherent recursive nature of the algorithm. We will deal with the latter type in another stage.

4.3.1 Dealing With Aging Buffers

Aging Buffers are a common idiom in many digital filters, that mimicks the behavior of a shift register. This programming pattern can be easily detected and efficiently implemented by High-Level Synthesis Compilers. In the IIR example, buffers `x` and `y` are aging buffers: at the end of each loop iteration, their values are shifted by one position.

Since the mechanism is exactly the same for buffer `y`, we will only focus on buffer `x`. Note that at iteration n, the value written in `x[1]` is the value of `x[0]` that was just read read from the input stream `in`. The value written in `x[2]` is the value copied into `x[1]` at iteration $n-1$, and so on. This copy cycle is materialized by a loop in the PRDG (the self-loop on x_9) and represented in Alpha by a self-referencing equation (see Figure 4.7):

This equation contains three branches. The first branch corresponds to the copy of the value read during the same iteration. The second branch corresponds to the copy of a value shifted during the previous iteration. The last one corresponds to the copy of the initial values of the buffer ($n = 1$, first iteration of the outer loop).

Notice that this equation is self-referencing because of the second branch. Ultimately, all the values copied in this branch either come from the input stream or from the initial

[1] http://maxima.sourceforge.net

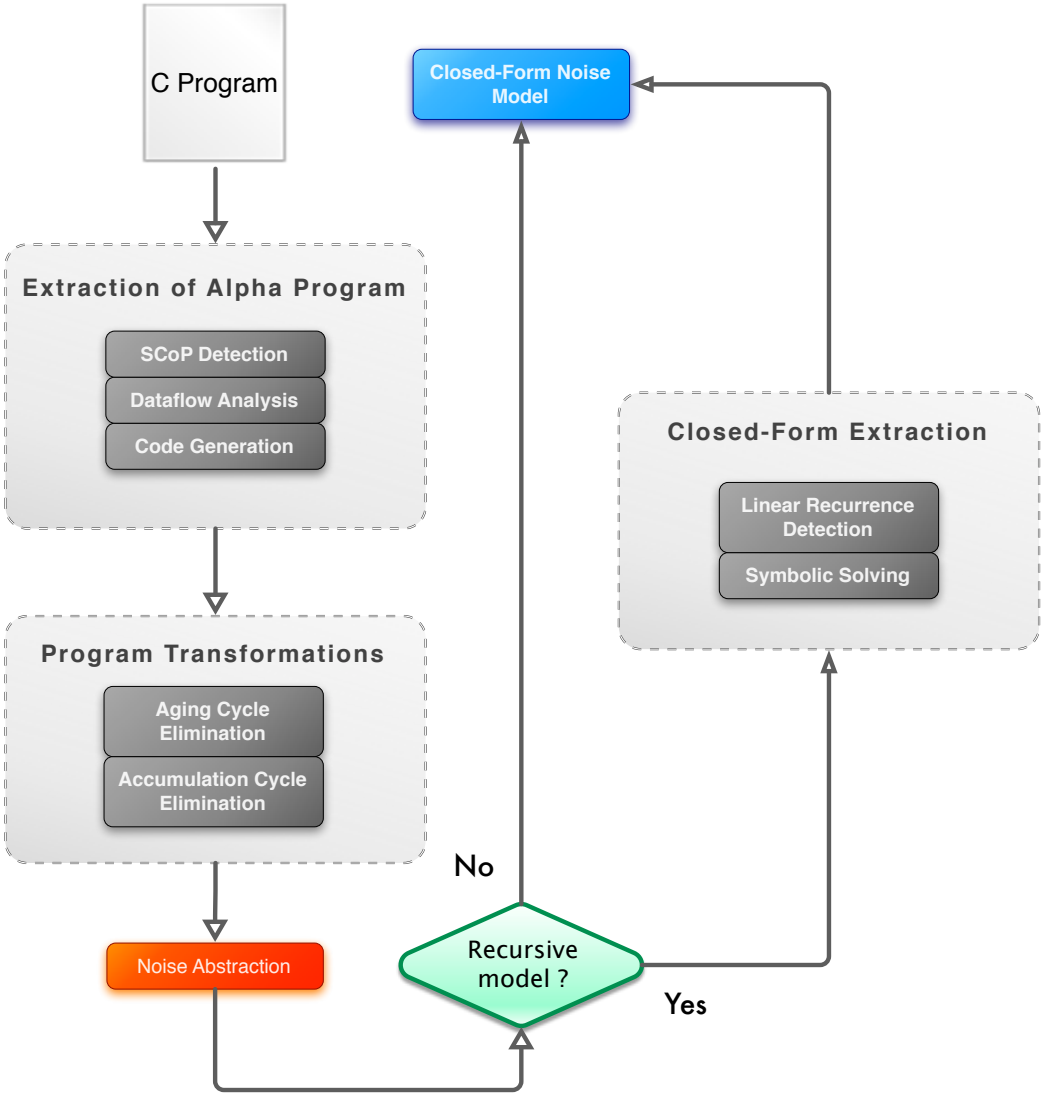


Figure 4.4: Overview of our approach

values of the buffer. To eliminate the self-reference, we need to compute the original source of the value for every point in the equation domain.

Equation `x_9` follows a simple pattern: it performs no computation. Each branch of the case statement defines a map to another value. This map is an affine function from the domain of `x_9` to the domain of the referenced equation. We can represent the whole equation as relation, by taking the union of the maps defined by each branch of the case. This copy relation can be represented graphically as a PRDG (see Figure 4.8). The self-reference is clearly apparent as a loop on x_9.

How can we remove the self-reference ? The right tool here is to compute the *transitive closure* of the relation. Recall that if $\rightarrow$ is a relation, $\rightarrow$ is transitive if $a \rightarrow b$ and $b \rightarrow c$ imply $a \rightarrow c$. The transitive closure of a relation $\rightarrow$ is the minimum transitive relation containing $\rightarrow$. It always exists and is unique. Since any value of buffer `x` either comes

```
for (n=1; n<INF; n++) {
  // Read a value from the input stream.
  x[0] = in[n];

  // Accumulation.
  accum1=0;
  for (i=0; i<N; i++) {
    tmp1 = x[i] * b[i];
    accum1 += tmp1;
  }

  // Accumulation.
  accum2=0;
  for (i=1; i<N; i++) {
    tmp2 = y[i] * a[i];
    accum2 += tmp2;
  }

  // Compute the result for this iteration
  // and write it to the output sream.
  y[0] = accum1 - accum2;
  out[n] = y[0];

  // Buffer aging
  for (i=0; i<N-1; i++) {
    x[N-i-1] = x[N-i-2];
    y[N-i-1] = y[N-i-2];
  }
}
```

Figure 4.5: C implementation of an IIR filter

from the input stream or from the initial values of `x`, the transitive closure of our copy relation must contain the origin of the value.

It is not always possible to compute the transitive closure of a polyhedral relation: the problem is undecidable. Fortunately, we are dealing with simple dependency functions: projections on a dimension and addition of a constant vector. In this case, finding the transitive closure is most of the time possible.

The closure maps each point in the domain to the set of points "visited" by the value. This set can contain several points in the domain of `x_9`, indicating that the value was shifted several times. Ultimately, each point in the domain is mapped to a single point

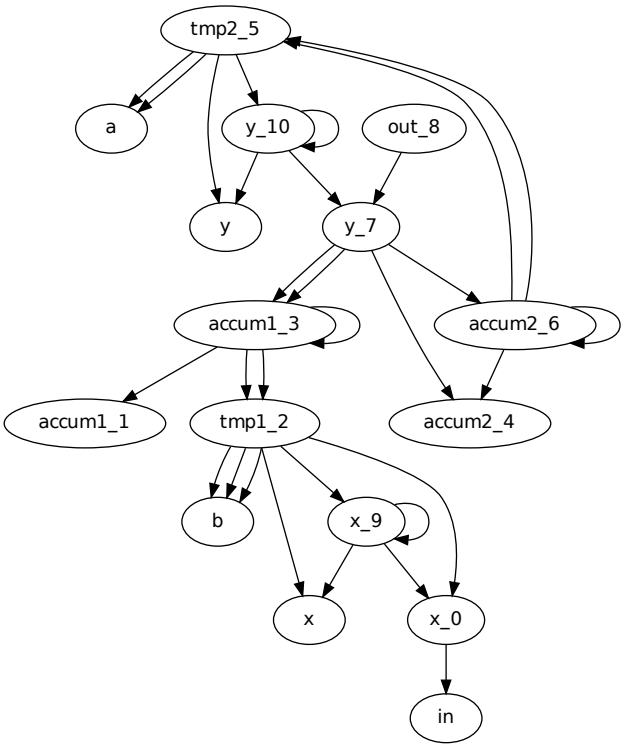
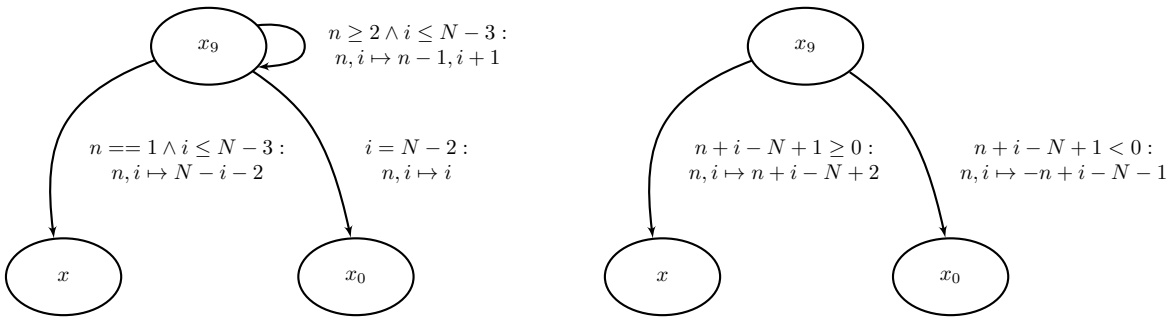


Figure 4.6: PRDG of the IIR filter, automatically extracted from the program in Figure 4.5. To improve readability, dependency functions were not represented. Nodes suffixed by an underscore followed by a number represent statements. They are named after the array or variable they modify, and the number indicates their position in the program. Non-suffixed nodes represent inputs (`in`, `a`, `b`) and initial values of the buffers (`x`, `y`).

from either `x` (the initial values of the buffer) or `x_0` (a read from the input stream). We

```
x_9[n,i] = case
  // Last input value.
  { | i-N+2==0 } : x_0[n] ;
  // Previous copy cycle
  { | n-2>=0 && -i+N-3>=0 } : x_9[n-1,i+1] ;
  // First iteration.
  { | n-1==0 && -i+N-3>=0 } : x[N-i-2] ;
esac;
```

Figure 4.7: Alpha equation for aging buffer



(a) Before

(b) After

Figure 4.8: Copy relation before and after cycle elimination.

simply eliminate the parts of the transitive closure that relate points in the domain of `x_9` and obtain a function giving, for each point in the domain, the origin of the value. The new Alpha equation is:

```
x_9[n,i] = case
  // Initial value.
  { | n+i-N+1 >= 0 } : x[n+i-N+2] ;
  // Value from the input stream.
  { | n+i-N+1 <  0 } : x_0[-n+i-N-1] ;
esac;
```

4.3.2 Accumulations

Accumulations are another source of cycles in the PRDG. At each iteration, values in buffer `x` and `y` are multiplied by a coefficient and the sum of these products is computed by an accumulating loop. Once again, this accumulation translates into a self-referencing equation in the Alpha program:

```
// Initialization of the accumulator
accum1_1[n] = 0;
// Accumulation per se.
accum1_3[n,i] = case
  { | i==0 } : accum1_1[n] ;
  { | i-1>=0 } : accum1_3[n,i-1] ;
esac + tmp1_2[n,i];
```

Fortunately, Alpha features a special syntax for these kind of accumulations: *reductions*. Reductions can be defined as the aggregation of a set of values with an associative and commutative operator. In this case, the operator is of course the addition and the aggregated values are the values of the variable `tmp_1`. This can be written in Alpha as:

```
accum1_3_reduce[n] = reduce(+, (n,i->n), tmp1_2);
```

The only confusing part of this snippet is probably the *projection function*: `n,i -> n`. Its meaning is the following: two points in the domain of `tmp1_2` belong to the same reduction if they have the same image by the projection function. In other words, the kernel of the projection gives the direction of the accumulation. In this example, values are accumulated along the second dimension: for a given n, the result of the reduction is the sum of the `tmp1_2[n,i]`.

Detection of reductions in an imperative program is a well-studied topic. The difficulty comes from the fact that accumulations can be written in many different ways. Fortunately, most reductions follow a very simple pattern and can be detected by simple pattern-matching.

4.3.3 Recursivity

Notice that even after these transformations, the PRDG of our program is still not acyclic (see Figure 4.9). It can be explained by the inherent recursivity of the IIR Filter. Indeed, the last few output samples are used to compute the next ones. There is not much we can do about it for now, but we will see in section 4.5 that working on an abstraction of the program will allow us to overcome this problem.

4.4 Noise Model Computation

The noise model computation builds an abstraction of the program describing its quantization noise behavior. The result is a SARE giving, for each point in the domain of each equation, an estimation of quantization noise power.

Our construction is based on classical noise power propagation rules, already presented in subsection 3.1.3 and summarized in Table 4.1. Note that multiplication is only handled if one of the operands is a constant. Indeed, if both operands are signals, noise power directly depends on the power of these signals and their covariance, which cannot be easily determined. If some statistical information is available about the signals, an over-approximation may be possible. We chose not to handle this case in this work. The user is asked to specify which inputs must be considered as arrays of constant coefficients.

To every variable in the original program, except for coefficients, we associate an integer input indicating its *resolution*. This resolution is expressed as the base-2 logarithm of its scaling factor (*e.g.*, a value of -3 denotes a scaling factor of 2^{-3}).

Input values are affected by a floating-point to fixed-point quantization noise. If the scaling factor of the target format is 2^k, the round-off error is a uniformly distributed white noise [17] over an interval of length 2^k. From the formula for the variance of a uniform distribution, quantization noise power is then: $\frac{2^{2k}}{12}$.

When the result of a computation is stored in a variable, this storage may or may not trigger an internal quantization, depending on whether the scaling factor of that variable is small enough to store the result in full accuracy. For each branch of an equation, we recursively construct an Alpha expression of the logarithm of the result's scaling factor.

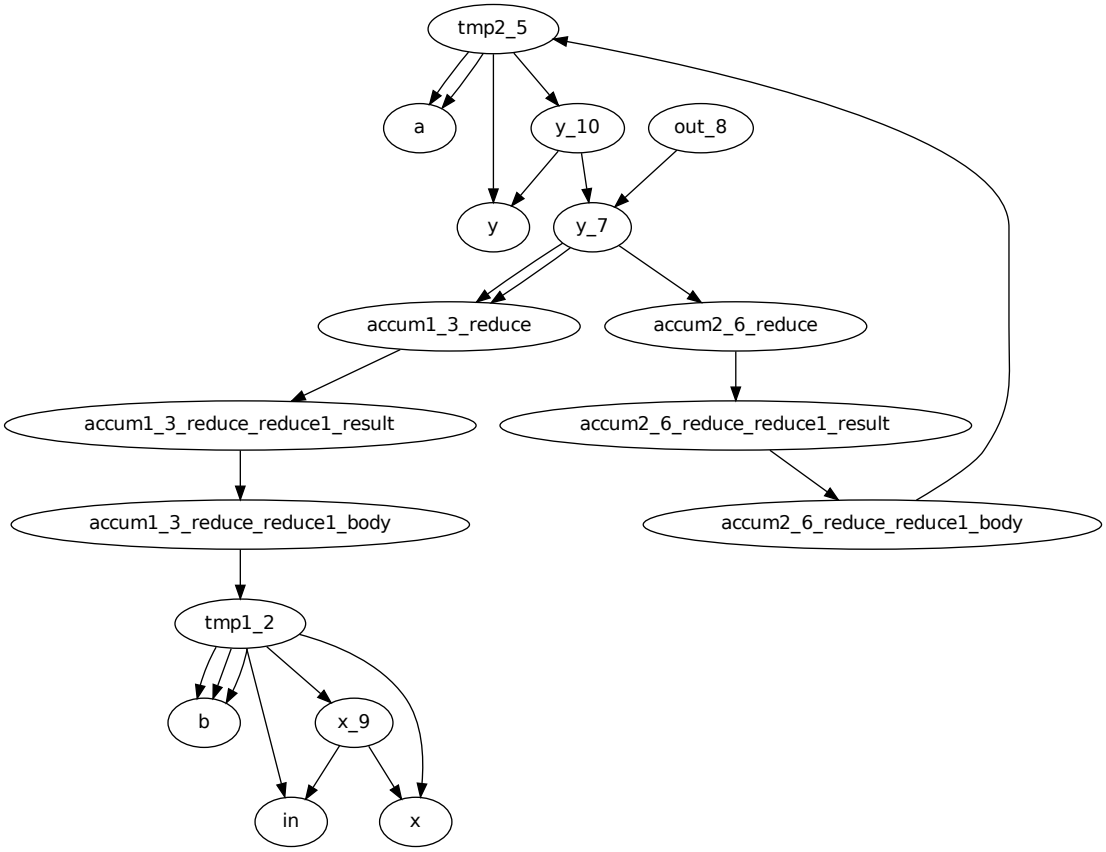


Figure 4.9: PRDG of the IIR filter after transformations

We compute the propagated noise power using the rules in Table 4.1. To this propagated noise, we add a conditional term corresponding to the local quantization noise. This term depends on the scaling factor of the variable and that of the result. If the resolution of the result is smaller, the corresponding quantization noise is given by the formula derived in [5].

Table 4.1: Noise power propagation rules

Operation	Noise Power	Remarks
$x + y$	$\sigma_x^2 + \sigma_y^2$	
$x - y$	$\sigma_x^2 + \sigma_y^2$	
$a \times x$	$a^2\sigma_x^2$	If a is a constant.

If an array initially contains only zeroes, storing one of these initial values (or its product with a coefficient) in a variable does not produce any quantization noise, no matter the resolution of that variable. The user can specify which buffers are zero-

initialized to obtain a more precise model.

Finally, we try to simplify the model by *inlining* equations that don't correspond to outputs of the sytem: we replace references to these equations by their definition. Note that it is not possible if the variable contains a self-reference. We distinguish three cases:

1. **The PRDG of the model is acyclic and contains only one output node.** This case corresponds to a single-output non-recursive filter, for which all PRDG cycles could be eliminated with the techniques presented in section 4.3. It is always possible to reduce such a model to a single equation, corresponding to the noise at the output of the system, with possibly more than one case branches.

2. **The PRDG of the model is acyclic and contains several output nodes.** This case corresponds to a multiple-output non-recursive filter. It is actually easy to treat this case, by considering each output node independently. We consider as many single-output filters as output nodes.

3. **The PRDG of the model contains cycles.** This case can happen if the filter is recursive or if some cycles could not be eliminated. It is not possible to treat this case in general. In the next section, a technique is proposed for a class of recursive filters.

4.5 Non-Recurrent Noise Models for Recursive Algorithms

In section 4.3, we managed to get rid of two kind of cycles in the PRDG of the IIR filter. Even after these transformations, a cycle remained (see Figure 4.9) because of the recursive nature of the filter. As a result, the noise model is also recursive.

The noise model we derive has the generic form of *linear recurrence with polynomial coefficients*. For ease of exposition, we will only consider input noise. We consider that each input is affected by a random noise of power σ^2. We obtain a simple model whose mathematical expression is:

$$P(n) = \begin{cases} b_0^2\sigma^2 & n = 1 \\ b_0^2\sigma^2 + \sum_{i=1}^{n-1} b_i^2\sigma^2 + \sum_{i=1}^{n-1} a_i^2 P(n-i) & n > 1 \wedge n < N \\ b_0^2\sigma^2 + \sum_{i=1}^{N-1} b_i^2\sigma^2 + \sum_{i=1}^{N-1} a_i^2 P(n-i) & n \geq N \end{cases}$$

The two first branches correspond the $N-1$ *initial terms* of the recurrence, that is, the first few values. The last branch corresponds to the *recurrence relation*.

In this example, we considered a parametric number of coefficients: N was unknown at design-time. In practice, the number (and value) of the coefficients is almost always known. For example, one may perform a convolution on a signal of variable-length, but the width of the convolution window is fixed during the algorithm exploration.

Suppose that $N = 3$. We can expand the model as follows:

$$P(n) = \begin{cases} b_0^2\sigma^2 & n = 1 \\ b_0^2\sigma^2 + b_1^2\sigma^2 + a_1 b_0^2\sigma^2 & n = 2 \\ b_0^2\sigma^2 + b_1^2\sigma^2 + b_2^2\sigma^2 + a_1 P(n-1) + a_2 P(n-2) & n \geq 3 \end{cases}$$

A generic algorithm to solve linear recurrences with polynomial coefficients in several computer algebra systems, such as MAXIMA or MATHEMATICA. Using MAXIMA, we obtain the following closed-form expression:

$$P(n) = \frac{\sigma^2 \left(-\frac{a_2}{a_1}\right)^{n-b_0^2\sigma^2} \left({b_2}^2 + {b_1}^2 + {b_0}^2\right)}{a_2 + a_1} - \frac{\sigma^2 \left({b_2}^2 + {b_1}^2 + {b_0}^2\right)}{a_2 + a_1} + b_0^2\sigma^2 \left(-\frac{a_2}{a_1}\right)^{n-b_0^2\sigma^2}.$$

This technique is only applicable for a very specific class of programs:

- One-dimensional recursive filters whose noise model is a linear recurrence with polynomial coefficients. It is the case of the IIR filter.
- Multi-dimensional recursive filters for which noise power only depends on the value of one dimension and can be transformed into a linear recurrence. It was the case of the stencil computation in chapter 3.

Indeed, noise models of multi-dimensional recursive filters usually take the form of a *multivariate recurrence*. These recurrences are notoriously hard to solve: as of today, no general solution is available, even for linear recurrences with constant coefficients [2].

4.6 Safety

It is easy to prove that unlike most classical static analyses, whose goal is to provide safe approximations of the semantics of a program, our technique is actually not safe.

Indeed, our analysis assumes that all quantization noises are uncorrelated. If x, y denote two signals and e_x, e_y the corresponding errors, the error associated to $x + y$ is simply $e_x + e_y$. The non-correlation assumption implies that

$$\mathrm{Var}(e_x + e_y) = \mathrm{Var}(e_x) + \mathrm{Var}(e_y),$$

and the noise power of the sum is the sum of the noise power for the operands.

Now, this formula is clearly wrong if x and y denote the same signal. Indeed, from the formula above, we have:

$$\mathrm{Var}(e_x + e_x) = 2\mathrm{Var}(e_x).$$

But from the properties of the variance, we know that:

$$\mathrm{Var}(e_x + e_x) = \mathrm{Var}(2e_x) = 4\mathrm{Var}(e_x).$$

It follows that, in some cases, our analysis provides overly optimistic estimations of noise power.

4.7 Implementation Status

Our flow has been partially implemented in the GECOS compiler infrastructure[2] developed in the CAIRN team at IRISA, and relies on the AlphaZ implementation of the ALPHA language.

More precisely, the following features have been implemented:

- Program transformations: cycle elimination for aging buffers and accumulations.
- Noise model computation.

These features have been tested on a variety of filters (FIR, IIR, image convolutions).

The following remains to be implemented:

- Post-abstraction model simplifications.
- Automatic recurrence detection.
- Recurrence solving with the help of a symbolic solver.

The first of these features does not present much difficulty. The two others are more subtle. Our goal is to implement model simplifications and recurrence detection by the end of the internship. It is unclear whether we will have time to start the work on symbolic solving.

[2] http://gecos.gforge.inria.fr

5 Conclusion and Future Work

In this work, we proposed a new method for accuracy analysis. This method is based on the polyhedral model and the SARE formalism. SAREs can concisely capture the semantics of a class of programs called *static control parts* (SCoPs).

We exposed two SARE transformations, whose goal is to reduce the number of cycles in the dependence graph. For non-recursive filters, these methods often result in an acyclic dependence graph, and a model of the quantization noise behavior of the program can easily be constructed. This model can be reduced to a single equation that can be used to perform wordlength optimization.

Recursive filters are more problematic: a quantization noise model can be constructed, but the recursivity is conserved by the model. We showed that in some specific cases, this model could actually be seen as *linear recurrence*. This recurrence can be solved into a non-recursive closed-form with a symbolic solver such as Maxima.

By construction, our technique is only applicable to SCoPs. In particular, it does not cover all *linear time-invariant* (LTI) systems. For example, it can handle the *finite impulse response* (FIR) and *infinite impulse response* (IIR) filters, but not the *fast Fourier transform* (FFT). Current analytical techniques can handle all LTI systems. It follows that our technique cannot yet be considered a full replacement of existing works. Also, finding a non-recursive closed-form solution of the noise model is not always possible, especially for multi-dimensional filters.

Several improvements to our technique can be considered. As of now, product between (non-constant) signals is not supported: the lack of information on the statistical distribution of the inputs and their correlation prevents us from providing any valuable information on the noise propagated by this operation. It may be possible to integrate such information to cover a larger class of programs.

No generic method exists to solve multivariate linear recurrences. Still, some of them can be translated to univariate recurrences by change of variable, and restricted classes are actually solvable. Solving these recurrences may allow us to compute a closed-form noise expression for more recursive filters.

Finally, we mentioned that our technique was not safe: it sometimes underestimates noise power, because of correlations between noises. It may be possible to detect of these correlations and provide safer approximations.

List of Figures

2.1 Anatomy of a floating-point format 8
2.2 Floating-point to fixed-point conversion 10
2.3 Example of a SFG 13
2.4 Noise model construction for a SFG. 14

3.1 Computation pattern of the stencil 15
3.2 Code for the weighted sum stencil. 16

4.1 Program, statement domains and dependencies 20
4.2 PRDG of the SCoP from Figure 4.1a 21
4.3 Excerpt of SARE for LU matrix decomposition 22
4.4 Overview of our approach 24
4.5 C implementation of an IIR filter 25
4.6 PRDG of the IIR filter 26
4.7 Alpha equation for aging buffer 26
4.8 Copy relation before and after cycle elimination. 27
4.9 PRDG of the IIR filter after transformations 29

List of Tables

2.1 Error propagation rules 11

3.1 Number of nodes of the SFG for the weighted sum stencil 17

4.1 Noise power propagation rules 29

Bibliography

[1] Bondhugula, U., Baskaran, M. M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., and Sadayappan, P. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Compiler Construction* (2008), L. J. Hendren, Ed., vol. 4959 of *Lecture Notes in Computer Science*, Springer, pp. 132–146.

[2] Bousquet-Mélou, M., and Petkovsek, M. Linear recurrences with constant coefficients: the multivariate case. *Discrete Mathematics 225*, 1-3 (2000), 51–75.

[3] Cachera, D., and Risset, T. Advances in bit width selection methodology. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors* (Washington, DC, USA, 2002), ASAP '02, IEEE Computer Society, pp. 381–.

[4] Cachera, D., Zegaoui, D., and Risset, T. Formal bit width determination for nested loop programs, 2002.

[5] Constantinides, G., Cheung, P. Y. K., and Luk, W. Truncation noise in fixed-point SFGs. *IEE Electronics Letters 35* (1999), 2012–2014.

[6] Constantinides, G. A. Perturbation analysis for word-length optimization. In *FCCM* (2003), IEEE Computer Society, pp. 81–90.

[7] Constantinides, G. A., and Woeginger, G. J. The complexity of multiple wordlength assignment. *Applied Mathematics Letters 15*, 2 (2002), 137–140.

[8] Feautrier, P. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming 20*, 1 (1991), 23–53.

[9] Lamport, L. The parallel execution of DO loops. *Communications of the ACM 17*, 2 (1974), 83–93.

[10] Le Verge, H., Mauras, C., and Quinton, P. The alpha language and its use for the design of systolic arrays. *The Journal of VLSI Signal Processing 3*, 3 (1991), 173–182.

[11] Menard, D., Rocher, R., and Sentieys, O. Analytical fixed-point accuracy evaluation in linear time-invariant systems. *IEEE Transactions on Circuits and Systems I: Regular Papers 55*, 10 (Nov. 2008), 3197–3208.

[12] Menard, D., and Sentieys, O. A methodology for evaluating the precision of fixed-point systems. *Acoustics, Speech, and Signal Processing* (2002), 3152–3155.

[13] Menard, D., and Sentieys, O. A methodology for evaluating the precision of fixed-point systems. In *ICASSP* (2002), IEEE, pp. 3152–3155.

[14] Moore, R. E. *Interval analysis*. 1966.

[15] Nguyen, H.-N., Ménard, D., and Sentieys, O. Novel algorithms for word-length optimization. In *Proc. 19th European Signal Processing Conference (EUSIPCO)* (Barcelona, Spain, Sept. 2011), pp. 1944–1948.

[16] Rocher, R., Menard, D., Scalart, P., and Sentieys, O. Analytical approach for numerical accuracy estimation of fixed-point systems based on smooth operations. *IEEE Trans. on Circuits and Systems 59-I*, 10 (2012), 2326–2339.

[17] Sripad, A., and Snyder, D. A necessary and sufficient condition for quantization errors to be uniform and white, 1977.

[18] T. Grötker, E. M., and Mauss, O. Evaluation of HW/SW tradeoffs using behavioral synthesis. In *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)* (Boston, oct 1996).

[19] Verdoolaege, S. *isl*: An integer set library for the polyhedral model. In *ICMS* (2010), K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds., vol. 6327 of *Lecture Notes in Computer Science*, Springer, pp. 299–302.

[20] Yuki, T., Basupalli, V., Gupta, G., Iooss, G., Kim, D., Pathan, T., Srinivasa, P., Zou, Y., and Rajopadhye, S. Alphaz: A system for analysis, transformation, and code generation in the polyhedral equational model. Tech. rep., CS-12-101, Colorado State University, 2012.

[21] Yuki, T., Gupta, G., Kim, D., Pathan, T., and Rajopadhye, S. Alphaz: A system for design space exploration in the polyhedral model. In *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing* (2012), LCPC '12.