An Infrastructure for Numeric Precision Control in the Ptolemy Environment

Seehyun Kim and Edward A. Lee
Department of Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720 USA

Abstract— **An abstract algorithm specification with idealized arithmetic must be made concrete with realistic arithmetic in the final phase of the "algorithm-to-implementation" design process in order to assess power consumption, hardware cost, and execution speed. In this paper, an infrastructure for refining an idealized model to get an architecture-dependent specification with finite precision is introduced. This infrastructure is built on top of the Ptolemy environment.**

I. Introduction

Dataflow is a powerful representation method for digital signal processing algorithms. One of its features is that it can effectively reveal concurrency implied in an algorithm by constructing a graph with nodes and arcs according to the data dependencies. Nodes carry out a certain computation using data tokens from input arcs, and emit new tokens through output arcs. As a design environment, Ptolemy supports dataflow programming for simulation and software synthesis. It also covers heterogeneous design which contains other computation models, such as discrete event, where graphs have different semantics.

An abstract algorithm specification with idealized arithmetic must be made concrete with realistic arithmetic in the final phase of the "algorithm-to-implementation" design process in order to assess power consumption, hardware cost, and execution speed. Especially, when retargetting the specification to general purpose fixed-point processors or reconfigurable processors, it's essential to find the optimum scaling and wordlength information to maximize the performance for given hardware cost.

In this paper, an infrastructure for refining an idealized model to get an architecture-dependent specification with finite precision is introduced. This infrastructure is built on top of the Ptolemy environment.

II. Fixed-point Models in Ptolemy

Ptolemy is an environment for simulation, prototyping, and software synthesis for heterogeneous systems. It uses object-oriented software technology to model each subsystem in a natural and efficient manner, and to integrate these subsystems into a whole. The kernel part has been written in C++. The basic unit of modularity in Ptolemy is the *block*, which is implemented with the class "`Block`." A block contains a module of code that is invoked at runtime, typically examining data present at its *input ports* and generating data on its *output ports*. The class "`PortHole`" is the base class for both input and output ports. Also a block can have some *states* so that it can preserve the current status until next execution. There are several classes for type-specific states, such as "`FloatState`."

A fixed-point system contains fixed-point blocks, which deal with data of finite precision. The fixed-point model of a block would have either fixed-point ports or fixed-point states. However, the fixed-point model may malfunction due to the finite wordlength effects. In order to prevent frequent overflows and excessive quantization noise, all ports and states of the fixed-point type have to be scaled properly and assigned with enough wordlengths.

The fixed-point data type is implemented in Ptolemy by the C++ class named "`Fix`". This class supports a binary representation of a finite precision number. In fixed-point notation, the partition between the integer part and the fractional part, which is the so-called binary point, lies at a fixed position in the bit pattern. Its position represents a trade-off between precision and range.

The "`Fix`" data type has some parameters concerning its interpretation. The total wordlength ("`twl`") indicates the length of the bit stream representing a value. The range of a variable is determined by the integer wordlength ("`iwl`") which is the length of the left-hand side bit stream to the binary point, while the fractional wordlength ("`fwl`") means the length of the remaining part and specifies the precision.

In addition, the sign format (2's complement or unsigned), precision reduction (rounding or truncation), and the overflow handling scheme are also parameterized.

III. Determination of Integer Wordlengths

Knowledge of the distribution is essential for estimating the range of a signal. It is easy to parameterize simple distributions, such as uniform, Gaussian, Laplacian, by a few statistical information. It is well known that speech signals have a Laplacian distribution. However, it is not possible to model all signals in practical systems by a simple distribution. They can be non symmetric or multimodal. Note that the estimated range of a multimodal signal could be very conservative or too small if we employ the rule for unimodal distributions. The scheme for estimating the range should be different according to the distribution. For instance, a Gaussian distribution can be covered by the four times of the standard deviation up to 99.99%. Thus, we can estimate the range of a simply distributed signal by means of the mean (μ) and the standard deviation (σ). For the multimodal situation, however, the standard deviation has a different effect on the range estimation process. This distribution requires $n{\cdot}\sigma$ to cover the distribution, where n is highly dependent on the distribution. Not only the mean and the standard deviation but the characteristics of the distribution are important for estimating the range as well [1].

After the execution of a system is completed, the mean, standard deviation, skewness, and kurtosis can be calculated using the statistics stored in each object. Then, the statistical ranges of ports and states are estimated as follows [2]: For unimodal and symmetric distributions, the range (R) can be effectively estimated by

$$R = |\mu| + n \cdot \sigma, \tag{1}$$

where n is proportional to κ, the kurtosis. Note the fact that for two symmetric distributions of an identical variance, the one of the larger kurtosis spreads more widely than the other. Thus, a larger n is needed for estimating the range of the signal with a larger kurtosis. Specifically, we use $n = k + 5$. Otherwise, the range is determined by the absolute maximum value during the execution.

Finally, integer wordlengths are obtained from their ranges, as shown in Eq. (2).

$$iwl_{min} = \lceil \log_2 R \rceil \tag{2}$$

where $\lceil x \rceil$ means the smallest integer larger than or equal to x. Note that the estimated integer wordlength of a signal might have to be modified according to its sensitivity to the performance of the whole system. For example, in the adaptation process of an LMS adaptive filter, the high precision of the error signal is crucial to the fast convergence and less steady state prediction error. The error signal might need more integer bits than the estimate for better performance.

A. New Feature of the Ptolemy Kernel

To record the statistics during the simulation, a few C++ classes have been developed.

- `RecordedObj` – base class. It keeps n-th order sum of past values, and provides a few methods to deal with statistical data.
- `RecordedFloatState` – inherited from `RecordedObj` and `FloatState`. This class collects statistics of an internal state of a fixed-point block.
- `RecordedInSDFPort` – inherited from `RecordedObj` and `InSDFPort`, which is the class for the input port in the SDF (Synchronous DataFlow) semantics.
- `RecordedOutSDFPort` – inherited from `RecordedObj` and `OutSDFPort`, which is the class for the output port in the SDF semantics.
- `SDFREStar` – inherited from `RecordedObj` and `SDFStar`, which is the base class for all primitive SDF blocks.

B. Generation of Models for Range Estimation

A range estimation model of the system can be generated by replacing blocks with range estimation blocks, which perform the identical functions and in addition, collect statistics during the execution.

An atomic block, which has no hierarchy, is described by the *Ptolemy preprocessor language* (ptlang) [3]. A range estimation block can be created by generating a new ptlang file from the original description in the ptlang specific editor provided in Tycho [4]. In this case users first have to select ports and states whose ranges are to be estimated. As an illustrative example, the ptlang description of a biquad filter is shown in Fig. 1. The figure also shows the dialog box querying the user to select ports and states to be examined. The ptlang description of a range estimation model of the biquad filter block is shown in Fig. 2. Note that the type of an input port, "`input`," and an output port, "`output`," was replaced with "`recordedfloat`," which used to be "`float`." The generated range estimation model can be loaded into the *Ptolemy interpreter* (ptcl) or the *interactive graphical interface* (pigi).

With range estimation blocks, a range estimation model can be generated from the original system graph in the ptcl specific editor. If the graph has been drawn in the *Vem* graphical editor, it can be easily converted to a ptcl script by `oct2ptcl`, a builtin program [5]. In order to generate a range estimation model, users first have to specify a range estimation block for each block to be replaced. Figure 3 shows a ptcl script for testing a simple biquad filter and a dialog box querying the user to specify an implementation for a block. In the query box a range estimation model, `MyBiquadRE`, has been selected for the biquad filter, `MyBiquad`. The ptcl script for a range estimation model is shown in Fig. 4. Note that the biquad filter block has been replaced with the range estimation block that was selected in Fig. 4.

C. Determination of Integer Wordlengths

The generated ptcl script of the range estimation model can be executed in ptcl. Then using a new ptcl command, `calcrange`, we can obtain estimated integer wordlengths as shown in Fig. 5. For the "`input`, we are suggested to assign 3 bits for the integer part.

IV. Generation of Fixed-point Models

A fixed-point model of the program graph is generated by replacing floating-point blocks with fixed-point equivalents. In case that the corresponding fixed-point block does not exist, it will be generated by substituting fixed-point variables for floating-point ones in the definition of the floating-point block. Figure 6 shows a fixed-point model of the biquad filter in Fig. 1. Note that estimated integer wordlengths are utilized in the fixed-point model; the new state variables, "`prec4_input`," specifies 3 integer bits and 14 fractional bits. Currently, the model uses 16 bits for representing the input signal. The optimal wordlengths, which minimize the hardware cost and satisfy the given performance criteria, can be found through iterative execution [6].

Due to the operator overloading capability of C++ we could obtain a bit-wise exact fixed-point block, in which all the operations including computation and assignment are conducted in the bit-accurate level, just by replacing the type of variables. Even if a fixed-point block exists already, it is possible to generate another functionally equivalent fixed-point block in order to model a different structure for computation. Thus, with architecture-specific fixed-point blocks we can construct a more precise fixed-point model of the program graph. Note that the finite wordlength effects are highly dependent on the implementation architecture. The ptcl script for a fixed-point simulation model is shown in Fig. 7. Note that the biquad filter block has been replaced with a fixed-point block, `MyBiquadFix`.

It is also possible to specify the precision for the signal on an arc. In this case all the data tokens passing on this arc would be reformatted to the given precision.

V. Concluding Remarks

An infrastructure for refining an idealized program graph with floating-point arithmetic into an architecture-dependent specification with finite precision has been constructed. This infrastructure would be helpful to further studies such as self-tuning systems.

References

[1] S. Kim, *A Study on the Fixed-point Implementation of Digital Signal Processing Algorithms*, Ph.D. thesis, Seoul National University, Feb. 1996.

[2] 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 I*, 1996, submitted.

[3] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Internatioal Journal of Computer Simulation, special issue on Simulation Software Development*, vol. 4, pp. 155–182, 1994.

[4] C. Hylands, E. A. Lee, and H. J. Reekie, "The Tycho user interface system," in *Proc. The 5th Annual Tcl/Tk Workshop '97*, Boston, Massachusetts, July 1997.

[5] University of California at Berkeley, *The Almagest: Vol.1 – Ptolemy 0.7 User's Manual*, 1997.

[6] Wonyong Sung and Ki-Il Kum, "Wordlength determination and scaling software for a signal flow block diagram," in *Proc. International Conference on Acoustics, Speech and Signal Processing*, Apr. 1994, vol. 2, pp. 457–460.

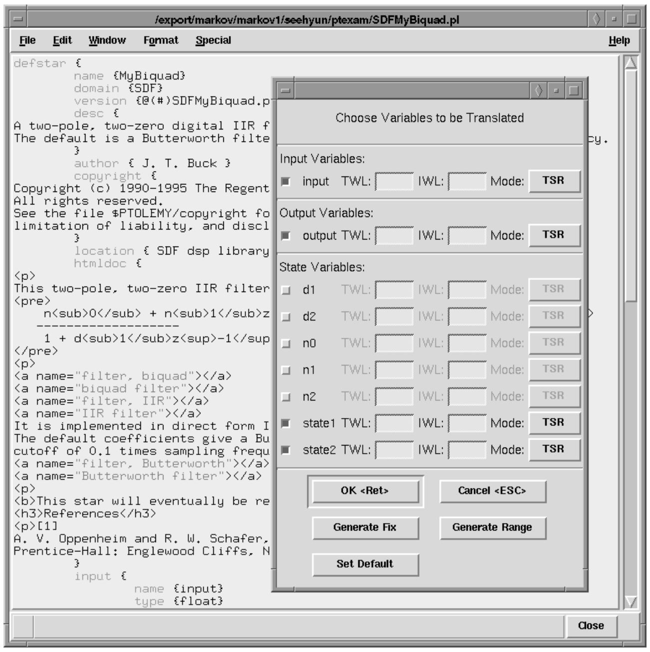

Fig. 1. A ptlang description for a biquad filter and a dialog box querying the user to select ports and states to be examined.

```
defstar {
    name { MyBiquadRE }
    ...

    input {
        name { input }
        type { recordedfloat }
    }
    output {
        name { output }
        type { recordedfloat }
    }
    defstate {
        name { d1 }
        type {  float  }
        default {  "-1.1430"  }
    }
    ...

    defstate {
        name { state1 }
        type { recordedfloat }
        default {  "0.0"  }
        desc {  internal state.  }
        attributes {  A_NONCONSTANT|A_NONSETTABLE  }
    }
    ...

}
```

Fig. 2. The ptlang description of a range estimation model of the biquad filter block.

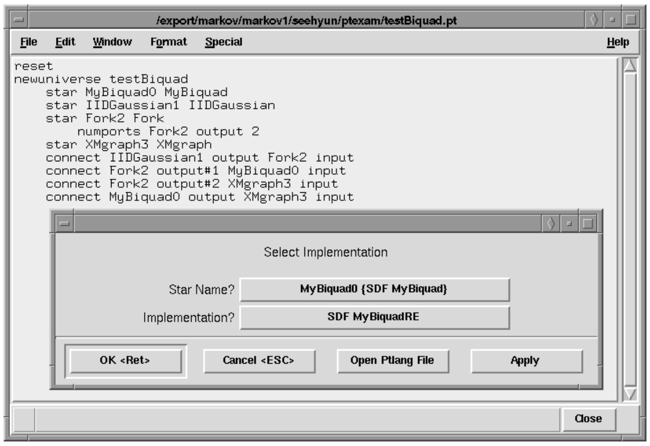

Fig. 3. A ptcl script for testing a simple biquad filter and a dialog box querying the user to specify an implementation for a block.

```
reset
newuniverse testBiquadRE
    star MyBiquad0 MyBiquadRE
    star IIDGaussian1 IIDGaussian
    star Fork2 Fork
        numports Fork2 output 2
    star XMgraph3 XMgraph
    connect IIDGaussian1 output Fork2 input
    connect Fork2 output#1 MyBiquad0 input
    connect Fork2 output#2 XMgraph3 input
    connect MyBiquad0 output XMgraph3 input
```

Fig. 4. The ptcl script for a range estimation model of the graph shown in Fig. 3.

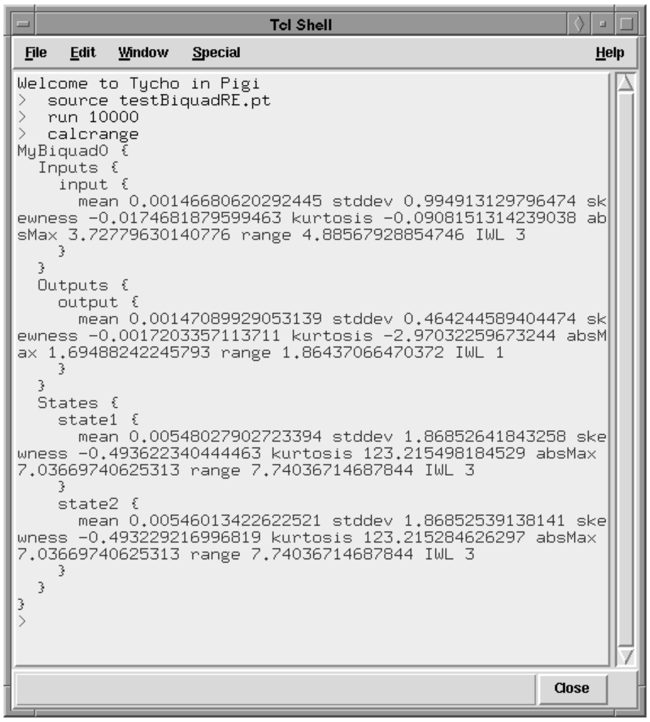

Fig. 5. Execution of the range estimation model and its result.

```
defstar {
    name { MyBiquadFix }
    ...

    input {
        name { input }
        type { fix }
    }
    output {
        name { output }
        type { fix }
    }
    ...

    defstate {
        name { state1 }
        type { fix }
        default {  "0.0"  }
        desc {  internal state.  }
        attributes {  A_NONCONSTANT|A_NONSETTABLE  }
    }
    ...

    go {
        Fix t = Fix (input%0) - Fix (d1) * Fix (state1) -
            Fix (d2) * Fix (state2);
        Fix o = t * Fix (n0) + Fix (state1) * Fix (n1) +
            Fix (state2) * Fix (n2);
        output%0 << o;
        state2 = Fix (state1);
        state1 = t;
    }
    ...

    defstate {
        name { prec4_input }
        type { string }
        default { "4.12,TSR" }
        desc { precision for input }
    }
    defstate {
        name { prec4_state1 }
        type { string }
        default { "4.12,TSR" }
        desc { precision for state1 }
    }
    ...

}
```

Fig. 6. The ptlang description of a fixed-point model of the biquad filter block.

```
reset
newuniverse testBiquadFix
    star MyBiquad0 MyBiquadFix
    star IIDGaussian1 IIDGaussian
    star Fork2 Fork
        numports Fork2 output 2
    star XMgraph3 XMgraph
    connect IIDGaussian1 output Fork2 input
    connect Fork2 output#1 MyBiquad0 input
    connect Fork2 output#2 XMgraph3 input
    connect MyBiquad0 output XMgraph3 input
```

Fig. 7. The ptcl script for a fixed-point model of the filter test graph shown in Fig. 3.