FILTER SYNTHESIS USING FINE-GRAIN DATA-FLOW GRAPHS

Waqas Akram, Cirrus Logic Inc., Austin, Texas

Abstract: This project is concerned with finding ways to synthesize hardware-efficient digital filters given technology and data rate constraints. The synthesis flow targets embedded systems implemented in application specific integrated circuits (ASICs). The flexibility inherent in such custom implementations provides opportunities for optimization down to the bit-level. This effort attempts to construct a convenient framework for the architectural manipulation of filter designs at the bit level, in order to reduce hardware complexity while meeting fixed data-rate constraints. The objective is to take an applicative language description of an algorithm and transform it into the most hardware-efficient descriptive language representation at the bit-level.

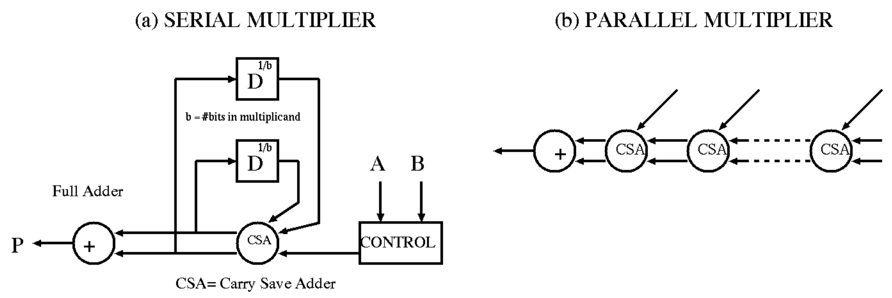
1.0 INTRODUCTION

Real-time signal processing systems implemented in application-specific integrated circuits (ASICs) have certain benefits, as well as unique limitations on performance. Depending on the application, area and power consumption can be two of the major concerns that drive an ASIC solution. The feasibility of portable embedded applications generally impose an upper limit on power consumption and cost. These limits translate directly to die area. The synthesis of such systems thus requires consideration of these properties. In contrast, embedded digital signal processor (DSP) targeted implementations consume more power and can occupy a larger die area. However, DSP implementations have the distinct advantage of programmability.

This project is concerned exclusively with the architectural synthesis of a subset of signal processing functions (digital filters) in ASIC hardware. The chief objective is to demonstrate that the use of fine-grain data flow graphs (FGDFGs) can improve the quality of results from a high-level synthesis tool, and can give a more accurate view of resource utilization. A great deal of research has been done in the area of high-level synthesis of these algorithms [1], and consequently, tools exist which perform this function with the minimization of resource usage at the forefront of consideration. Examples include HYPER [2,3] from the University of California at Berkeley, as well as proprietary tools from companies such as Altera, Synopsys, Cadence, etc. Due to it's relatively accessible documentation, HYPER will be used to compare performance wherever possible.

In any synthesis system, the algorithm is represented as a data flow graph (DFG) [1]. This representation enables the tool to efficiently and easily manipulate the system resource usage, while maintaining functionality. The first step is to ensure that real-time constraints are met. The most stringent of these constraints is the data rate. Real-time systems are characterized by an effective infinite loop of operations performed on incoming data, and require that all processing be done at the rate of arrival, in order to avoid build-up of data. High-level transformations, such as retiming [1,4], folding [1] and unfolding [1,5,6,7] can be applied, in order to meet the input data rate. These transformations typically treat low-level operations like addition and multiplication as monolithic entities, and attempt to satisfy the constraints using the most cost-effective implementation of these low-level operations. For example, HYPER uses a hardware library consisting of several implementations of multipliers. Each implementation displays superior performance in a particular subset of attributes such as area, speed, and power consumption. After the data rate constraint is met, a reduction in resource utilization is attempted. In other words, high-level transformations are again applied, this time to the data rate satisfied DFG, in order to produce the most hardware efficient architecture. HYPER performs these operations in several stages, the first of which is the "design space exploration" stage, which places bounds on the performance [9] of the algorithm given the hardware library. This stage relies on the library implementations of the low-level operations to remain static throughout the synthesis process. In other words, even though the tool is free to choose from the entire library until the end of the synthesis process, the actual number of choices remains static. Subsequent stages, like transformations for resource utilization [8], use the information gathered in the design space exploration stage to constrain the optimization process. Finally, the resultant structure of the DFG is used to choose the least expensive library units in the final hardware mapping stage.

FIGURE 1. Fine-grain data flow representations of a multiplier



2.0 FINE-GRAIN DATA FLOW

Fine-grain data flow refers to the representation of low-level operations using the same structure and semantics as the higher-level modules in the hierarchy. Thus, a multiplier may have serial and parallel fine-grain descriptions as shown in Figure 1. These fine-grain modules can be placed in a module library, where they can be accessed by the synthesis tool, similar to the way HYPER uses the hardware library. The difference here is that a hardware library, similar to that used in HYPER, is still required and is still accessed at similar points in the synthesis process.

Based on the design space exploration, the hierarchy is flattened using particular FGDFG representations of low-level operations, while applying transformations for resource utilization. The key difference between HYPER's "matching process" and the FGDFG approach is that in the former, the hardware units are fixed. That is, their sub-elements cannot be used in the transformation process. The units are selected based on the speed, area, and/or power trade-off, and then used as monolithic units in the final hardware mapping. In HYPER, the DFG transformation and resource allocation stages are distinct. However, the FGDFG approach merges the two stages in order to exploit hardware redundancies within the low-level elements. For example, a multiplier is essentially a series of additions, so a multiply-accumulate unit, and hence a finite duration impulse response (FIR) filter, can be implemented with a single adder and a number of registers (for storage of intermediate results), provided the real-time data rate constraint can be met. If this implementation of the multiplier (expressed as a FGDFG) is substituted into the high-level DFG, the transformation stage is free to arrive at this conclusion, if necessary. An alternate implementation of the multiplier, shown in Figure 1b, using carry-save adders (CSA), may be used to improve performance even further, since the speed penalty of carry propagation need not be suffered.

The major advantage of FGDFG transformations is the ability to tap into the variety and abundance of algorithms and architectures for implementing building-blocks (multiplication units, multiply-accumulate units, add-compare-select units, etc.), while obtaining the most resource-efficient implementation possible. In contrast, DSP targeted synthesis cannot exploit this advantage, because the granularity of operations is usually fixed by the target DSP architecture. In theory, the granularity of the ASIC targeted synthesis FGDFG representation can be made as fine as the gate-level.

3.0 Modeling and Implementation

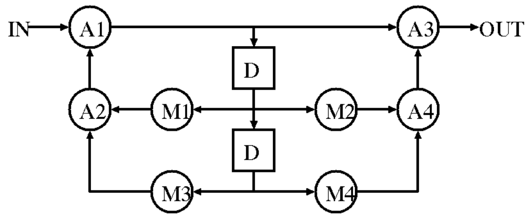
The most common computational model for signal processing systems is Synchronous Data Flow (SDF) [10]. This is a well-understood and mature model, and is ideally suited for data-intensive graphs. There are existing and proven transformation, scheduling, and resource allocation techniques for SDF. However, there is an implicit assumption of the bit-width of data flowing over arcs. This is not an issue for DSP targeted systems, which share this assumption, and rely on the system designer to set the target bit-width for all arcs, and thus the target embedded DSP. However, ASIC targets are very sensitive to the width of these arcs, and would greatly benefit, in the minimum resource usage sense, from a system which maintains just enough precision on each arc. Feed-forward systems are usually the majority of benefactors when this consideration is made. Thus, the bit-width of each arc needs to be maintained as attributes of each arc. One could argue that an eight-bit output from a block can simply be modeled as an eight-token output arc (according to SDF semantics and nomenclature). This however, constrains the output to a fixed width. Certain transformations may require a different precision to be maintained when the block is placed elsewhere. The solution is to model the eight-bit output as eight outputs in a homogeneous SDF graph. For simplicity, DFGs initially use constant bit-widths for arcs (with a "bit-width" attribute on each arc). There is also a "delay" attribute with each node.

In the transformation stage, the objective function [9] has to be modified in order to account for the granularity. In this attempt, the total area is used as a simple objective. This is calculated by adding the area attributes of each element (including delay-elements) currently in the FGDFG, and a global variable is maintained throughout the transformation process. Because today's standard-cell ASICs are implemented in multi-layer interconnect processes, interconnect usage reduction has a relatively lower priority.

Due to the enormous complexity of implementing such a synthesis system, a subset of transformations has been implemented, including retiming, some limited associativity, and folding. In some cases, it is assumed that the DFG has been optimally-unfolded [5] in order to meet the data rate constraint. The goal here is not to implement efficient algorithms for manipulating data flow graphs, but to demonstrate that solutions obtained from using fine-grain data flow graphs instead of coarse-grain data flow graphs are of higher quality, and in some cases, are the only solutions that exist for certain real-time constraints. This analysis is not intended to favor one transformation over another: both coarse-grain and fine-grain approaches are tested using the limited set of transformations implemented. However, the retiming transformation is a case where the fine-grain approach benefits by a greater amount.

Synthesis consists of several major steps. First, the input DFG is scheduled, and the resource utilization bounds are calculated [3]. A heuristic is used to substitute more “retiming friendly” fine-grain models of elements in the critical path(s). The retiming friendliness attribute on FGDFG modules is purely subjective, and is a ratio of the granularity of the particular fine-grain module to the granularity of the surrounding graph in the top-level graph. This is the module allocation stage. All arcs with delays are then broken, and a directed acyclic graph is constructed. The graph is scheduled, and resources are allocated. This allocation is used as a bound for subsequent transformation optimization. The graph is transformed (using retiming, associativity, etc.), and re-scheduled to compute the new resource allocation. A branch-and-bound scheme is used to find local minima, while back-tracking on the transformation decision tree. Once the local minima is found, the module allocation step is repeated with slightly different combinations of fine-grain modules, and the new local minima is computed in this way. After a user-selected number of module combinations, the list of local minima are compared, and the best solution selected. The synthesis produces a data flow graph with allocated resources, as well as a few instantiated control blocks. These are simply firing schedules for the surrounding modules, and can be synthesized into multiplexors. The resource complexity of these blocks is estimated as being proportionate to the length of their [periodic] schedules.

FIGURE 2. Initial Biquad filter



4.0 Results

As a simple example, Figure 2 shows a second-order biquad section. Assuming the adder executes in 1 time unit, and the multiplier executes in 2 time units, the coarse-grain biquad requires two multipliers and one adder to execute in 4 time units, as shown in Table 1.

FIGURE 3. Retimed fine-grain Biquad filter

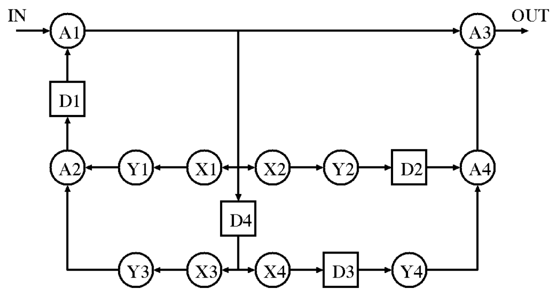


Figure 3 shows the same biquad, but with fine-grain multiplier model from Figure 1b, where the carry save array/tree has been grouped into X (partial product reduction stage) and the 2-input carry-propagate adder is called Y. This implementation requires only one [effective] multiplier (since, 1 X + 1 Y = 1 MULT), and one adder to execute in 4 time units, as shown in Table 1.

Both implementations use five registers. The effective values are used, since the adder “shares the tasks” of the multiplier, thereby improving the resource utilization for a single multiplier solution. A seventh order infinite duration impulse response (IIR) filter implemented in the same manner would require three multipliers less than the coarse-grain solution, while still meeting the data rate constraints. Using finer-grain graphs for the multiplier and the adders, and more accurate timing models, a hardware resource reduction of about 32% was achieved.

TABLE 1. Comparison between coarse-grain and fine-grain approach using simple biquad decomposition

COARSE-GRAIN BIQUAD					FINE-GRAIN BIQUAD			
TIME UNIT	**ADDER**	**MULT1**	**MULT2**		**TIME UNIT**	**ADD1**	**ADD2**	**Reduced MULT**
1	A2	M2	M3		1	A1	Y4	X3
2	A1				2	A4	Y3	X1
3	A4	M1	M4		3	A3	Y1	X2
4	A3				4	A2	Y2	X4

This comparison is by no means comprehensive, but demonstrates that fine-grain data flow modeling can improve the quality of resource usage minimization tools. Coupled with the extensive transformation library that current coarse-grain synthesis tools employ, a very powerful tool can emerge. Given more time, it would be interesting to implement a full suite of transformations, and use existing benchmark graphs for an accurate comparison of the actual speedup. Of course, the search for a solution is largely dependent upon the heuristic used to select fine-grain modules from the module library in the first place.

5.0 REFERENCES

[1] K.K. Parhi, “Algorithm Transformations for Concurrent Processors,” *Proceedings of the IEEE*, vol. 77, no. 12, pp. 1879-1895, December 1989.

[2] C. Chu, et. al., “HYPER : An Interactive Synthesis Environment for High Performance Real TIme Applications,” *Proc. IEEE ICCD Conference*, November 1989.

[3] J. Rabaey and M. Potkonjak, “Resource Driven Synthesis in the HYPER System,” *IEEE ISCAS 1990*, vol. 4, pp.2592-2595, New Orleans, LA, May 1990.

[4] C. E. Leiserson, F. Rose, and J. Saxe, “Optimizing Synchronous Circuitry for Retiming,” *Caltech Conference on VLSI*, pp. 87-116, March 1983.

[5] K.K. Parhi and D.G. Messerschmitt, “Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding,” *IEEE Transactions on Computers*, vol. 40, pp. 178-195, February 1991.

[6] L.-F. Jeng and L.-G Chen, “Rate-Optimal DSP Synthesis by Pipeline and Minimum Unfolding,” *IEEE Transactions on VLSI Systems*, vol. 2, pp. 81-88, March 1994.

[7] L.-F. Chao and E. Sha, “Retiming and Unfolding Data-Flow Graphs,” *Proc. of IEEE International Conference on Parallel Processing*, part II, pp. 33-40, August 1992.

[8] M. Potkonjak and J. Rabaey, “Optimizing Resource Utilization Using Transformations,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 277-292, March 1994.

[9] J. Rabaey and M. Potkonjak, “Estimating Implementation Bounds for Real Time DSP Application Specific Circuits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 669-683, June 1994.

[10] E.A. Lee and D.G. Messerschmitt, “Synchronous Data Flow: Describing DSP Algorithms for Parallel Computation,” chapter in *VLSI Signal Processing II*, S-Y Kung, Editor, IEEE Press, New York, 1986.