SYSTEMATIC EXPLOITATION OF DATA PARALLELISM IN HARDWARE SYNTHESIS OF DSP APPLICATIONS[1]

Mainak Sen and Shuvra S. Bhattacharyya

{mainak,ssb}@eng.umd.edu

Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies,
University of Maryland, College Park, 20742, USA.

ABSTRACT

In this paper, we describe an approach that we explored for low-power synthesis and optimization of digital signal, image, and video processing (DSP) applications. In particular, we consider the systematic exploitation of data parallelism across the operations of an application dataflow graph when synthesizing a dedicated hardware implementation. Data parallelism occurs commonly in DSP applications, and provides flexible opportunities to increase throughput or lower power consumption. Exploiting this parallelism in dedicated hardware implementation comes at the expense of increased resource requirements, which must be balanced carefully when applying the technique in a design tool. We propose a high level synthesis algorithm to determine the data parallelism factor for each computation, and based on the area and performance trade-off curve, design an efficient hardware representation of the dataflow graph. For performance estimation, our approach uses a cyclostatic dataflow intermediate representation of the hardware structure under synthesis. We then apply an automatic hardware generation framework to build the actual circuit.

1. MOTIVATION

High-level synthesis has been of primary importance in the field of DSP as area and power considerations are critical in the DSP domain. Design space exploration can be done effectively from a high level description as some inherent traits are more obvious in the high level abstraction and become obscure in the low level implementations. Dataflow has proven to be an attractive high-level computation model for programming DSP applications. A restricted version of dataflow, termed synchronous dataflow (SDF), that offers strong compile-time predictability properties, has been studied extensively in the DSP context [1][6]. We have developed an algorithm and Verilog code generation framework for optimal application of data-parallel hardware implementations to SDF graphs. Further, since $power \propto V^2 f$, where V is the operating voltage and f is the operating frequency, we can reduce both V and f, thus trade performance for lower dynamic power consumption. As an example, we consider a simple 3-tap FIR filter. Figure 1 shows a synchronous dataflow graph representation of such a filter.

Here, the inputs to each module consume one unit of data upon each invocation, and the modules produce one unit of data at the output. From this SDF graph representation of the filter, we can clearly see that data parallelism through replication of hardware blocks can be used for each of the modules to increase the throughput. The given dataflow graph provides enough information to derive a hardware implementation of the filter. But by analyzing the given dataflow, we can increase the throughput of the circuit by duplicating the multipliers and creating parallel datapaths to them. Figure 2 shows the 3-tap FIR filter of Figure 1 with data-parallel factors of $\langle 2, 2, 3 \rangle$ and $\langle 1, 1 \rangle$ for the multipliers and the adders respectively. The additional switches needed for sending data to multiple instances of the modules are also shown. This possibility of configuring a data-parallel hardware implementation results in a wide design space to probe around in order to maximize throughput or minimize power consumption.

The rest of this paper is structured as follows. In Section 2 we present the formal problem statement as well as the optimality of the solution provided by the algorithm proposed. Section 3 provides the framework used for automatic hardware code generation from the optimally configured circuit given by our algorithm. Section 4 provides results for some typical DSP subsystems and implications of those results. Section 5 draws a conclusion of our work and provides some useful directions for further exploration.

2. PROBLEM STATEMENT AND SYNTHESIS ALGORITHM

In this section we present the formal statement of the synthesis problem that we address, and present the algorithm developed to solve it. We also show that the algorithm has polynomial complexity and provides optimal synthesis results.

2.1 Problem statement

In this model, each functional module M (dataflow graph vertex) that has a data-parallel implementation is characterized by an overhead factor, denoted v_M, which approximates the amount of additional functional resource area (or cost) required for each level of data-parallel implementation. Specifically, an m-level data-parallel implementation of M (an implementation with m parallel copies of the hardware block) is modeled as requiring a functional resource cost of

$$A_M + (m-1)v_M A_M = A_M(1 + (m-1)v_M), \qquad (1)$$

where A_M is the cost of a single instance of module M (without application of the data parallelism transformation). Similarly, a

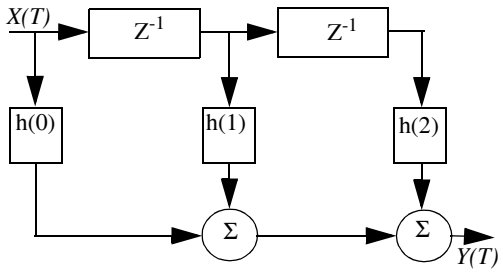

Figure 1. An SDF graph representation of a 3-tap FIR filter with production and consumption rates uniformly equal to one.

1. This research was supported by the Advanced Sensors Collaborative Technology Alliance.

module-independent area or cost switching overhead is used to model the switching area or communication cost required for the connection between an incoming (outgoing) data stream and an m-way parallel network of hardware modules operating at m times the data rate of the stream. Under this formulation, the data parallelism synthesis problem becomes one of determining a strategic mapping $\mu : V \rightarrow Z+$, where V denotes the set of application modules (dataflow graph vertices), $Z+$ denotes the set of positive integers, and $\mu(M)$ denotes the data parallelism implementation level (the number of parallel instantiations) of module M. So, we are concerned with the constraints of area (cost), power consumption, and throughput, and the objective of data parallelism synthesis is to achieve an optimal or near-optimal configuration μ that targets the relevant constraints and optimization criteria across these metrics.

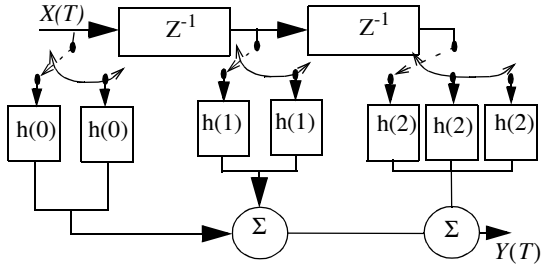

Figure 2. The 3-tap FIR filter shown in Figure 1 with different data-parallel factors for the different multipliers and adders. Datapaths and their implementation with switches are shown.

2.2 Proposed Algorithmic Solution

The algorithm follows a greedy approach. At every iteration, it checks for the module which when duplicated gives the maximum performance benefit. The algorithm terminates when duplicating any hardware module violates the area constraint.

Performance benefit is measured by the function 'Performance Analysis' in Figure 3. The switching characteristics of any circuit is very aptly represented by the dataflow computational model known as Cyclo-static dataflow(CSDF) [2]. In this model, a module can have different phases in which it can consume and produce data at different rates. The initial dataflow along with the data-parallel factors and switches can now be effectively represented by an equivalent CSDF graph. Performance of the resulting dataflow graph is measured by its throughput. This is done by first forming the equivalent Homogeneous SDF (HSDF) graph of the CSDF graph [2]. An HSDF graph is an SDF graph whose data production and consumption rates per firing are uniformly equal to one. Every module in the CSDF graph forms a cycle C in which the different elements in one cycle corresponds to different phases of the CSDF graph in its most simplified form. Let $W(v_i)$ be the execution time of each of the modules in one such cycle C_j. Then $\Sigma W(v_i)$ is the total weight of the cycle C_j. The mean cycle weight of cycle C_j in an HSDF graph is defined as

$$(\Sigma W(C_j))/(Delay(C_j)) \quad . \qquad (2)$$

[7] where $Delay(C_j)$ is the total number of delay elements in C_j. The cycle with the maximum mean cycle weight is called the *critical cycle*; it gives the maximum achievable throughput for the graph.

Let T_i be the execution time for a module i and let the data-parallel factor for this module be N_i. Then the throughput

Data Structures Used:

```
list = queue of structures;
newlist = queue of structures;
structures = struct {
module_info;
number_of_copies_of_the_module;
}
```

Main algorithm:-

```
Form the newlist by enqueueing all the modules;
while(newlist_not_empty) {
    list = newlist;
    while(list_not_empty) {
            Module_i = dequeue from the list;
            m = (Module_i —>copies) ++;
            A_new = v_i A_i + Area_old;

            if (A_new < Area available) {
                    Performance Analysis(Module_i);
            }
    }

    get the module with **best** result;
    (Module_i —>copies) ++ in newlist;
}
```

Performance Analysis:-

```
Form the corresponding Cyclostatic Dataflow(CSDF);
CSDF to Homogeneous Synchronous Dataflow(HSDF);
Maximum Cycle Mean (MCM) Analysis;
Store the result;
Enqueue_newlist(Module_i, copies);
```

Figure 3. The algorithm used to get the data-parallel factors for each module.

for module i is $(N_i/T_i) = P_i$. The throughput of the entire system is thus $Min(P_i)$ over all i. Also let B_i be the base area of i. Our objective is to maximize $Min(P_i)$ subject to the constraint $\Sigma f(N_i, B_i) \leq A_{max}$ where A_{max} is the maximum die-area on the chip available for hardware implementation. $f(N_i, B_i)$ is approximately $N_i B_i$ if the overhead for multiple hardware units is negligible.

In the greedy approach taken, we repeatedly select the bottleneck module i and increase its data-parallel factor by one, provided area constraint is not violated. In effect, we expand the module i just enough so that it is no longer the bottleneck for the system. This greedy procedure results in optimal configurations; this can be seen from the following argument. If a module i that is the bottleneck has a current data-parallel factor p_i, and only a data-parallel factor of $p_i + k$ or more will remove it from being the bottleneck, then the algorithm will always choose i for the next $(k-1)$ iterations (provided there is enough area). In other words, the algorithm always devotes available area toward improving the bottleneck module, which is the best that can be done under a given area constraint. Improving the performance of any non-bottleneck task cannot improve the overall throughput. The maximum number of times a module i is visited by the algorithm is $(A_{max})/B_i$ which is polynomial in the number of modules. Thus the overall complexity of the algorithm is polynomial.

3. AUTOMATIC VERILOG CODE GENERATION

After the synthesis algorithm provides the data-parallel factor vector, we simulate the actual hardware. For that we have developed an automatic Verilog code generation framework that is built on top of Ptolemy II [4], a design environment for modeling and design of heterogeneous embedded systems.

3.1 Motivation for code generation

To measure the effectiveness of our algorithm, we have performed area and power calculations on a number of circuits, which are presented in the results section. We synthesized the data-flow graphs in hardware to verify our results from the algorithm. Thus our results are backed by hardware synthesis rather than software simulation.

3.2 Code generation methodologies

We have explored two different approaches for code generation. We either describe the Verilog code for a module as a congregation of functions it performs or we have a standard code library that implements the basic structure for that module. The only difference is in the granularity in which we confront the code generation problem.

The two different approaches were considered based upon flexibility and speed for code generation. If the user needs more customized code generation, then the functional description approach is more suited to his needs. But the user should have a sound knowledge on synthesizable code generation for the generated code to work correctly. As for code generation from the standard library, the user need not know the intricacies of code generation. A basic parameterized framework for a particular module is already provided in the library, the user needs to invoke it with the required parameters, one of them being the number of inputs to the module. For example, an adder can be a two bit adder, or any n-bit adder and this parameter needs to be specified at the time of invocation. This is a very reasonable approach for code generation, and also we can generate area optimized code that is suitable for low power applications as the library modules are optimized. Overall, the library approach is easier and usually produces better code. We discuss this approach in more detail in the following section.

3.3 Library approach to code generation

From the input SDF graph, we extract all the modules needed for code generation. The only way to have a one-to-one correspondence between the module and the correct code from the library is to use a uniform nomenclature. For this purpose, we have used the intuitive names such as adder, delay, multiplier, etc. for the corresponding modules. After the modules are identified, we import the module definition from the library. The different modules are wired after the wiring pattern is extracted from the input SDF graph. Evidently, the wires are the edges in the graph. If the data-parallel factor for a particular module is n, then the code for it is defined only once but instantiated n times. We add a switch to manage the data parallelization for the n instantiations.The generated code for the above mentioned 3-tap FIR is given in Figure 4. The adder module is the only complete module.The input, output and reg statements are omitted from the other module definitions for brevity.

The code generated is divided into synthesizable and verifiable parts. This feature is maintained by using the testbench approach for Verilog code generation. We generate two separate files, one file contains code for the system being designed, and the other contains the test generator and the monitor. As a result, the first file contains the synthesizable part and the second file contains wires to input and output modules needed for verification of the circuit. This approach is described in detail in [8]. Figure 4 shows only the synthesizable code for a 3-tap filter.

We also generate the code for a switch when we simulate the hardware for the graph shown in Figure 2. A simple $1 \rightarrow 2$ switch generated by our code generator is shown in Figure 5.

```
module adder(in1, in2, in3, out, clk);
    input [15:0] in1;
    input [15:0] in2;
    input [15:0] in3;
    input clk;
    output [15:0] out;
    reg [15:0] out;
   always @(posedge clk) begin
        out <= in1 + in2 + in3;
   end

endmodule

module multiplier(in1, in2, out, clk);
   always @(posedge clk) begin
        out <= in1 * in2;
   end
endmodule

module delay(in1, out, reset, clk);
   always @(clk or reset) begin
   if (reset == 1) begin
            out <= 0;
   end
   else if (clk == 1) begin
            out <= in1;
   end
    end
endmodule

module top(in, clk, reset, out);
   assign param0 = 'h0;
   assign param1 = 'h1;
   assign param2 = 'h2;

   adder a(w2, w4, w6, out, clk);
   multiplier m1(in, param0, w2, clk);
   multiplier m2(w3, param1, w4, clk);
   multiplier m3(w5, param2, w6, clk);
   delay d1(in, w3, reset, clk);
   delay d2(w3, w5, reset, clk);
endmodule
```

Figure 4. Generated synthesizable Verilog code for the 3-tap FIR filter described in Figure 1.

```
module   switch(in1,   datainready1,   in2,
datainready2, reset, clk, dataoutready, out);
    always @(posedge clk) begin
       if (reset == 1) begin
               counter <= 0;
               datainready1 <= 0;
               datainready2 <= 0;
               dataoutready <= 0;
               end
       else if(counter == 0) begin
               counter <= counter + 1;
               out <= in1;
               dataoutready <= 0;
               datainready1 <= 1;
               datainready2 <= 0;
               end
       else if(counter == 1) begin
               counter <= counter + 1;
               out <= in2;
               dataoutready <= 1;
               datainready2 <= 1;
               datainready1 <= 0;
               end
       end
    endmodule
```

Figure 5. The example Verilog code of a simple $1 \rightarrow 2$ switch.

4. RESULTS

We evaluated our algorithm on a number of typical DSP subsystems. We present the results of three such subsystems. The first one is a cascade of a simple adder (input node) and multiplier (output node). Simulation results from Synopsys Design Compiler [10] are shown in Table 1. The second circuit is a 3-tap FIR circuit shown in Figure 1. Third is a second order IIR filter. We observe that the data-parallel factors provided by our synthesis algorithm are supported by the data values produced by Design Compiler.

Table 1. Results of adder multiplier circuit from Synopsys

Config	Area (μcm^2)	Dynamic Power(mW)	Critical Time(ns)
$M = \langle 1 \rangle$ $A = \langle 1 \rangle$	27394	1.52	10.31
$M = \langle 1 \rangle$ $A = \langle 2 \rangle$	33269	1.71	9.51
$M = \langle 2 \rangle$ $A = \langle 1 \rangle$	51903	2.83	5.93

Table 2. Results of 3-tap FIR filter from Synopsys

Config	Area (μcm^2)	Dynamic Power(mW)	Critical Time(ns)
$M_{1,2,3} = \langle 1, 1, 1 \rangle$ $A_{1,2} = \langle 1, 1 \rangle$	68632	3.89	10.68
$M_{1,2,3} = \langle 1, 2, 1 \rangle$ $A_{1,2} = \langle 1, 1 \rangle$	88266	4.62	10.68
$M_{1,2,3} = \langle 2, 2, 2 \rangle$ $A_{1,2} = \langle 1, 1 \rangle$	126634	6.08	5.88

Table 3. Results of a second order IIR filter from Synopsys

Config	Area (μcm^2)	Dynamic Power (mW)	Critical Time (ns)
$M_{1,2,3,4} = \langle 1, 1, 1, 1 \rangle$ $A_1 = \langle 1 \rangle$	134812	1.9	7.63
$M_{1,2,3,4} = \langle 2, 2, 2, 2 \rangle$ $A_1 = \langle 1 \rangle$	259309	3.5	4.09

The library used for synthesis is for 0.25μ CMOS logic process. For the first circuit, our algorithm suggested $M = \langle 2 \rangle, A = \langle 1 \rangle$. The synthesized circuit gives maximum throughput for the same configuration under an area constraint of 60000 μcm^2. For the 3-tap FIR filter, the best performance is provided by $M_{1,2,3} = \langle 2, 2, 2 \rangle$, $A_{1,2} = \langle 1, 1 \rangle$ which tallies with the output of our algorithm when $A_{max} = 130000 \mu cm^2$. The second row of Table 2 shows that even though the multiplier is the bottleneck, providing only one parallel datapath to one of the multipliers does not decrease the critical path time, and accordingly, our algorithm does not choose this as an improved configuration. Even though the results shown here are for moderate-sized graphs with numbers of modules on the order of tens, the core algorithm is of low polynomial complexity, and therefore our approach can be expected to scale efficiently to larger systems.

5. CONCLUSION

The above tables show some of the possible configurations of the mentioned dataflow graphs that do not violate the given area constraints. It can be observed that in all of the above cases, the data-parallel configuration suggested by our synthesis algorithm was the solution with the best performance.

Data parallelism for DSP hardware implementation is a well-known concept [9]; the contribution of our paper is in the full vertical integration of data-parallelism-based transformations with synchronous dataflow graph analysis, cyclostatic dataflow-based performance analysis, synthesizable Verilog code generation, and hardware synthesis using the Synopsys Design Compiler. This integration provides a fully automated design flow that produces optimal exploitation of data parallelism for SDF-based designs.

Useful directions for further work include hardware synthesis from more general dataflow models, such as integer-controlled dataflow [3], and well-behaved dataflow [5]; and systematic integration with other flowgraph transformations for multi-objective synthesis.

6. REFERENCES

[1] S. S. Bhattacharyya, P. K. Murthy and E. A. Lee. Software Synthesis from Dataflow Graphs. *Kluwer Academic Publishers,* 1996.

[2] G Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Dataflow. *IEEE Transactions on Signal Processing*. Vol 44, No 2, February 1996.

[3] J. T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control systems. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 508-513, October 1994

[4] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE,* January 2003.

[5] G. R. Gao, R. Govindarajan, and P. Panangaden. Well-behaved programs for DSP computation. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing,* March 1992.

[6] E. Lee and D. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, pages 55–64, September 1987.

[7] S. Sriram and S. S. Bhattacharyya, Embedded Multiprocessors: Scheduling and Synchronization, *Marcel Dekker Inc.* 2000.

[8] .D. E. Thomas, P. Moorby. The Verilog Hardware Description Language. *Kluwer Academic Publisher,* Nowell Massachusetts 1991.

[9] M. Williamson. Synthesis of parallel Hardware Implementations from Synchronous Dataflow Graph Specifications. *PhD thesis, Department of EECS, University of California at Berkeley,* May 1998.

[10] Synopsys Design Compiler User Manual, *Synopsys*.