

Using behavioral clustering to improve quality of results for DSP designs

David J. Pursley
Forte Design Systems
1501 Reedsdale Street #302
Pittsburgh, PA 15233
+1-412-321-4350 x15
pursley@forteds.com

Introduction

This paper introduces behavioral clustering, a technique that significantly improves performance, area, and power consumption of digital signal processing (DSP) designs.

Behavioral clustering is a methodology for combining and optimizing hardware processes in such a way that the quality of results (QoR) is significantly better than when the processes are implemented separately. Because DSP designs are normally specified as a set of communicating processes, behavioral clustering is easily leveraged on these applications.

By applying these techniques to real-world designs, we have found that behavioral clustering can improve performance by over 85% for some designs, while decreasing area by as much as 18% for others. In most cases, both performance and area can be improved by applying behavioral clustering.

What is behavioral clustering?

Clustering has been previously used as a way to partition a design between chips, datapaths, or even hardware and software. In that case, the starting point is typically a large monolithic behavioral design, and the goal is to keep tightly communicating blocks together. This minimizes the QoR degradation caused by the communication overhead of partitioned processes.

DSP applications, however, have a different starting point. Normally, they are specified as a set of communicating processes, with each process corresponding to a DSP building block (filter, up/down-converter, FFT, etc.). Traditionally,

each block is implemented separately as register-transfer level (RTL) code, and the RTL blocks are then combined to form the overall DSP design.

On the other hand, as shown in Figure 1, behavioral clustering can be used to combine the building block processes into a smaller set of larger processes before they are implemented in RTL. These large processes are then implemented via a standard hardware design flow. By intelligently choosing the processes to combine into a single process, we can create designs that have better performance, area, and power characteristics than when created with the traditional flow.

The improvement in QoR is a function of several techniques that are applied to the combined processes. These include latency hiding, functional unit reuse, datapath compilation, and communication reduction. While these techniques would be difficult to apply to a set of RTL processes, they can be quickly and automatically applied to behavioral processes through commercial behavioral synthesis tools.

The following sections discuss previous related work before going into a detailed explanation of the advantages of behavioral clustering. We explain the application of this technique in light of different optimization objectives, and follow-up with case studies from real-world designs. The paper concludes with ideas for future work in this area.

Previous work

As mentioned above, the closest work to this space had to do with partitioning one process into

sub-processes [Lagnese91][Vahid99]. The goal was usually to minimize the impact of the communication overhead partitioned processes require. The starting point for the partitioning problem is traditionally a “large” algorithm.

Behavioral clustering can be thought of as the inverse of partitioning, applied to a different target application. In DSP, Image Processing, etc., the system-level algorithm is typically very naturally decomposed into sub-processes (filters, up/down-converters, FFT’s, etc.). By clustering, or combining, the sub-processes into larger processes, both area and performance can be significantly improved.

Behavioral clustering

Behavioral clustering combines and optimizes hardware processes in such a way that the quality of results (QoR) is significantly better than when the processes are implemented separately. (We’ll define a process as a datapath and state machine. Multiple processes have multiple state machines.) A simple example of this is shown in Figure 1.

In order to discuss this in more detail, we’ll use the following notations.

P_a denotes process a.

P_b denotes process b.

$P_{a \cup b}$ denotes the clustered process.

$Area(P_a)$ denotes the silicon area of P_a . A higher number means a larger area design.

$Latency(P_a)$ denotes the time to fully complete of P_a . A higher latency means the process takes more time to complete (worse performance).

$Throughput(P_a)$ denotes the data rate of P_a , the rate at which data is produced or consumed. A higher throughput means a faster data rate (better performance).

Basically, behavioral clustering allows the designer or synthesis tool to work with larger block sizes in order to leverage optimizations across block boundaries. These optimizations include latency hiding, functional unit reuse, datapath optimization, and communication reduction.

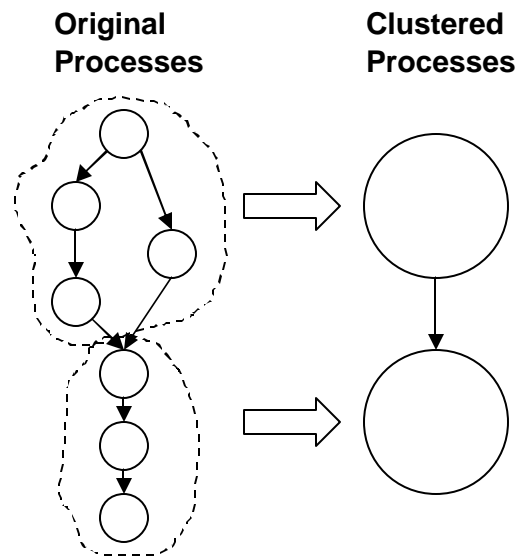


Figure 1: Simple behavioral clustering

In short, behavioral clustering attempts to meet one or more of the following goals.

Improve area:

$$Area(P_{a \cup b}) < Area(P_a) + Area(P_b)$$

Improve latency:

$$Latency(P_{a \cup b}) < Latency(P_a) + Latency(P_b)$$

Improve throughput:

$$Throughput(P_{a \cup b}) > Throughput(P_a) \text{ AND}$$

$$Throughput(P_{a \cup b}) > Throughput(P_b)$$

Which goal(s) are most important is dependent of the specific design and application.

Limitations

There are three main limitations on how to group processes into clusters. These have to do with the practical issues of clocking, communication, and scalability.

For practical purposes, datapaths and state machines are generally clocked off a single clock. That is, while there may be multiple clock signals in a given design, any given block only uses one directly. Therefore, it is logical to impose the restriction that processes to be clustered must be in the same clock domain.

Second, behavioral clustering appears to have the most impact on processes that are directly communicating with each other. As explained below, three of the four optimizations enabled by behavioral clustering are only applicable to communicating processes. Therefore, we’ll restrict clusters to contain only communicating processes.

The third limitation is the practical limitation of scalability. Note that the resulting block sizes are by definition larger than what the RTL designer normally works with, which may impact the overall design cycle and consequently time-to-market. Because of that, this technique is most naturally applied when using a behavioral synthesis tool. Some behavioral synthesis tools also have the advantage of automatically applying these optimizations.

The following sections will describe the optimizations behavioral clustering enables.

Latency hiding

Latency hiding is a performance optimization that reorders operations such that some of the computations of two processes happen at the same time, reducing the total number of clock cycles required by the calculation. This is shown in Figure 2.

The goal of this optimization is to reduce overall latency. That is:

$$Latency(P_{a \cup b}) < Latency(P_a) + Latency(P_b)$$

Because this optimization involves starting operations from P_b while P_a is still executing, this transformation applies mainly to sequential communication processes.

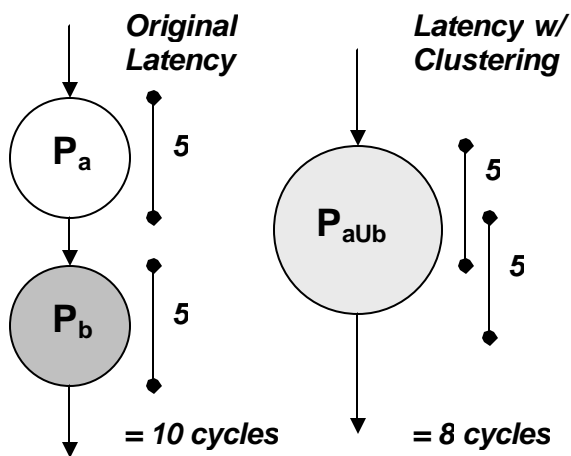


Figure 2: Latency hiding

Functional unit reuse

Functional unit reuse is an area optimization where computation in P_a shares some functional units (adders, multipliers, etc.) with P_b once the processes are clustered. This is shown in Figure 3.

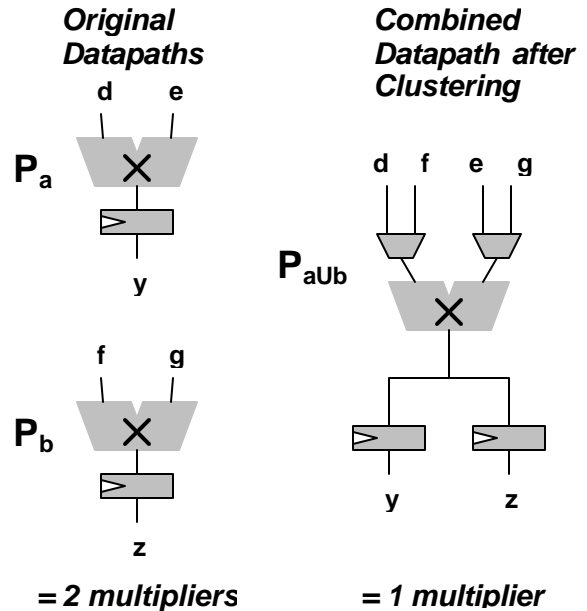


Figure 3: Functional unit reuse

The goal of this optimization is to reduce area. That is:

$$Area(P_{a \cup b}) < Area(P_a) + Area(P_b)$$

Note that this implies that the functional units to be shared are not fully utilized. For example, consider P_a and P_b , both using 24-bit multipliers. Functional unit reuse will be able to use one multiplier only if the sum of the utilization factors (percent time the multiplier is active) of the two multipliers is less than 100%.

A quick rule of thumb to determine if functional unit reuse is even possible is if each process has a throughput less than 1. In general, fully pipelined (throughput=1) designs do not allow for any sharing at all, since all functional units may be active every cycle.

Datapath compilation

Datapath compilation can be an area or performance (or both) optimization that takes significant portions of the computation from the

datapath and creates a highly-optimized gate-level component using a datapath synthesis engine.

The goal of this optimization can be either:

$$Latency(P_{a \cup b}) < Latency(P_a) + Latency(P_b)$$

$$Area(P_{a \cup b}) < Area(P_a) + Area(P_b)$$

In some cases, both goals can be realized.

Datapath synthesis is commonly used in RTL design. The general premise of datapath synthesis is that an equation or sequence of operations is specified, as well as a standard cell technology library, and the datapath synthesis engine creates a highly optimized datapath component implementing the sequence of operations [Synopsys99]. The components can be optimized for area, speed, and/or power, and often the datapath synthesis engine can optimize the geometry for floorplanning and possibly factor in physical synthesis parameters. A very high-level view of datapath synthesis is shown in Figure 4.

The power of datapath synthesis can be leveraged by behavioral clustering in two ways.

First, it can create large optimized datapath components that could be used by both blocks, and then reuse them via the functional unit reuse optimization. This can improve performance by using highly optimized components while also reducing area by reusing them.

Second, if performance is the only goal, very large pieces of the algorithm can be given to the datapath synthesis engine, assuming it can accept it. These pieces may be very large and span the processes being clustered. This highly optimized component will often significantly increase the performance of the design.

The second application of datapath compilation is applicable only to communicating processes, since the input to datapath synthesis engines is an “atomic” sequence of operations.

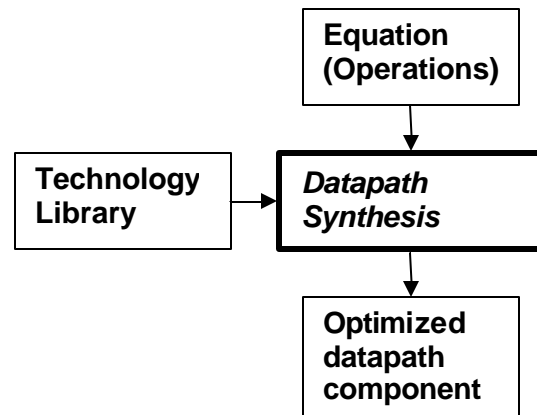


Figure 4: Datapath synthesis

Communication reduction

Communication reduction is the result of removing any need for communication mechanisms between the processes that were clustered. Communication could be via buffers, memories, handshakes, system buses, etc. By combining the state machine and datapath of the two communication processes, this overhead is no longer necessary.

The goal of this optimization is often a combination of all performance goals:

$$Latency(P_{a \cup b}) < Latency(P_a) + Latency(P_b)$$

$$Throughput(P_{a \cup b}) > Throughput(P_a)$$

$$Throughput(P_{a \cup b}) > Throughput(P_b)$$

For example, combining processes could remove a system bus transaction, thereby improving both latency and throughput. Similarly, any type of handshaking may cost at least a clock cycle to resolve, impacting latency.

The most complex case of this is when clustering the two processes can result in a change to the memory architecture. For example, in image processing applications, clustering processes in certain ways can minimize the amount of storage required throughout the system.

Because this transformation has so much impact, it is commonly done today. Communication mechanisms, and specifically their overhead, are usually carefully considered at the architectural level. In fact, this is usually the only reason

behavioral clustering is done, although it may not be thought of as such. While not novel in itself, this transformation is included for completeness.

Applying behavioral clustering

Applying the above optimizations is a very difficult task to do in RTL (and the designer would likely only have one chance to get it right due to time-to-market pressures), but it is perfect for behavioral synthesis. Applying behavioral clustering to help determine the best micro-architecture is simply something that cannot be done in today's traditional RTL flow.

Behavioral synthesis has been around for years, and there are several excellent texts on the subject [Thomas90][Elliott00]. An excellent quick introduction to behavioral synthesis is also available [Meredith94].

Case studies

In the examples below, the designer clustered the block-level design before using Forte Design Systems' Cynthesizer™ behavioral synthesis product [Forte04] to exploit the optimization opportunities presented by behavioral clustering.

The starting point for each design was a SystemC [Black04] or C++ design.

MP3 case study

MP3 decoders are widely understood, so I won't go into detail here. More information on the MP3 codec can be found in [Pan95].

The MP3 decoding algorithm was downloaded from the web [AnsLab00] and the sub-band synthesis and hybrid filter bank (inverse MDCT's) processes were modified for synthesis. The target performance for this design was 44,100 samples/sec.

The processes were synthesized separately, giving a total area of 1064k sq.um including memories using the TSMC 0.18um process. Then the two processes were clustered together in order to attempt to decrease silicon area. The results are shown in Table 1.

Implementation	Performance	Area (sq.um)
Original	44.1ksamp/sec	1064k
Clustered processes	44.1ksamp/sec	975k

Table 1: Comparison of MP3 decoder implementations

Note that the area reduction is actually more significant than it first appears. Both implementations had 558k in memory, so excluding memory the area savings was 17.6%.

The area reduction was accomplished by the tool applying the datapath optimization and functional unit reuse optimizations. Basically, by starting with a larger process, more opportunities for sharing were found.

Image pipeline case study

This customer example originally contained six sequential processes creating an imaging pipeline. The exact algorithm is proprietary, but it contained standard image processing blocks such as color conversion, horizontal and vertical filters, etc.

All but one of the processes had been designed for a previous project, but it was clear that they would not meet the desired performance goal of 66 frames per second at the highest resolution, 720x576 (PAL format). (Later experiments showed that the best performance achievable with separate processes was under 40 frames per second at the target resolution.)

All six processes were clustered to one, and behavioral synthesis was used on the clustered process. A range of RTL implementations were created, ranging from approximately 50 to over 75 frames per second. Interestingly, the area for the fastest implementation was less than 15% more than the original separate process version, even though performance had nearly doubled. This is summarized in Table 2.

Implementation	Performance	Area (relative)
Original	40 fps	1.00
Clustered processes	75 fps	1.15

Table 2: Comparison of image processing implementations

The reason higher performance could be attained with only minimal area impact is a combination of latency hiding and functional unit reuse. Before

clustering, each of the processes was running as fast as possible, which means they were using a lot of area. When the processes were clustered, it turned out that some of the processes could run much slower due to latency hiding. This, in turn, reduced utilization of functional units, making functional unit reuse a real possibility.

The performance saved by removing communication overhead was minimal due to the structure of the design, and datapath compilation was not used on this example.

Conclusion

Above, we presented behavioral clustering as a novel approach to solving the problems of increasing designs sizes and demanding QoR requirements. This approach is perfectly suited to DSP designs, as the starting point is a set of communicating processes. By applying this technique and leveraging existing synthesis tools, significant improvements in both performance and area can be had.

There are several directions for future work in this area. First, these techniques should be attempted on processes that do not communicate. While the impact will likely be lesser, it could still be quite significant. It is also possible that other optimizations will be discovered during that investigation. Further work could automate the application of behavioral clustering.

Finally, the cost functions of existing partitioning algorithms could be augmented to account for lost optimization opportunities. It is now clear that the cost of using separate processes is more than just the cost of communication.

References

[AnsLab00] AnsLab, *Ans_MP3_Decoder*, <http://www.anslab.co.kr/>, 2000.

[Black04] Black, David C. and J. Donovan, *SystemC: From the Ground Up*. Kluwer Academic Publishers, 2004.

[Elliot00] Elliot, John P., *Understanding Behavioral Synthesis: A Practical Guide to High-Level Design*. Kluwer Academic Publishers, 2000.

[Forte04] Forte Design Systems, *Cynthesizer™ User's Guide*. <http://www.ForteDS.com/>, 2004.

[Lagnese91] Lagnese, E.D. and D. E. Thomas, "Architectural Partitioning for System Level Synthesis of Integrated Circuits," *IEEE Transactions on Computer Aided Design*, v. 10, July 1991.

[Meredith04] Meredith, Michael, "A look inside behavioral synthesis," *EEDesign*, April 8, 2004.

[Pan95] Pan, Davis, "A tutorial on MPEG/Audio Compression," *IEEE Multimedia Journal*, Summer 1995.

[Synopsys99] Synopsys, Inc., *Module Compiler Datasheet*. 1999.

[Thomas90] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, and R. L. Blackburn. *Algorithmic and Register Transfer Level Synthesis: The System Architect's Workbench*. Kluwer Academic Publishers, 1990.

[Vahid99] Vahid, Frank. "Techniques for Minimizing and Balancing I/O During Functional Partitioning". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, January 1999.