Bounded Bundled Data

Willem Mallon
Asynchronous Research Center
Portland State University
w.c.mallon@gmail.com

Abstract

Bundled data channels provide a control signal often called "request" to confirm validity of data on a bundle of data wires. To make the request certify data validity, the traditional bundling convention insists that data precede the request signal.

The generalization of the bundling convention outlined here permits any calculable timing relationship between data and request. I call this generalization "Bounded Bundled Data"(BBD).

When the data precedes the request signal, the request signal can be used to generate a data capture signal. When the request signal precedes the data signal, the request signal can be used to set up the appropriate multiplexers on the data path.

By relaxing the bundled data convention, we create a framework that allows the designer to balance data and control delays on computation paths. It also allows CAD tools to avoid unnecessary synchronization. Avoiding unnecessary synchronization reduces delay insertion in CAD.

BBD also offers an abstraction to reason about control and data delay. The abstraction allows the computation of control overhead on critical paths, and can be used to optimize those paths. The insight offered by the abstraction allows the designer to reorganize his high-level design for better performance.

Keywords: self-timed designs, asynchronous circuits, CAD, dataflow networks.

1 Introduction[1]

In this paper I formalize the *Bounded Bundled Data*(*BBD*) convention. *BBD* is similar to bundled data in that it connects precisely one control and one data valid event, but in the *BBD* convention, the relation in time between the two events varies throughout the design.

At various point in a design, data must be sampled or stored. This requires synchronization of control and data events. When the bundled data convention is used, synchronization occurs *always* on each data path. Using the *BBD* convention, we can postpone synchronization until it is absolutely necessary, and reduce the amount of delay insertion required by taking the entire control path into account. This reduces significantly the amount of control delay insertion required.

The *BBD* convention requires global static timing analysis, but this analysis can be implemented using the standard static timing tools. Furthermore, the analysis divides into a global part that follows the design structure, and a local part that requires analyses of short timing paths.

The *BBD* convention offers a powerful abstraction of components when analyzing their data-delay and control-overhead behavior within the design, and for frequently used paths of the design.

I present an analysis method that is simple and effective. It also appears suitable for automatic optimization. The kind of optimizations that are enabled by *BBD* are complementary to optimizations such as slack-matching [2] or canopy analysis [4].

[1]This is ARC report ARC2011-WM-09. This document contains information developed at the Asynchronous Research Center at Portland State University. You may disclose this information to whomever you please. You may reproduce this document for any not-for-profit-purpose. Reproduction for sale is strictly forbidden without written consent by the author. Copies of the material must contain this notice

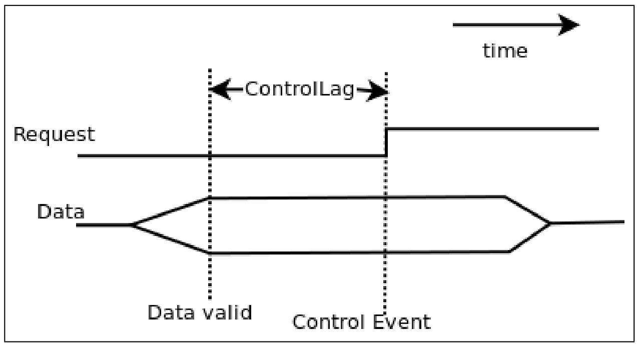


Figure 1 The bundled data convention states that the control event occurs *after* the data have become valid. This is appropriate if the control event creates a data capture signal

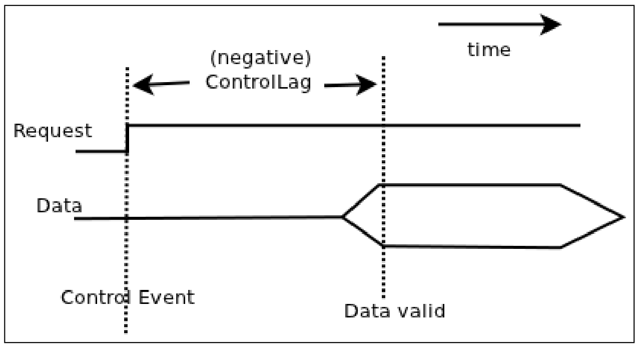


Figure 2 The *Bounded Bundled Data* convention allows the control event to precede the data valid event. Static timing analysis can compute a bound for the difference in time between the two events.

The idea to have a looser bundled data convention is not new. Various forms have been reported, e.g. [7]. My contribution formalizes and extends the concept. This allows a complete implementation in CAD tools. This is in contrast to the approach of [7], that uses ad-hoc approximations. The implementation of *BBD* presented here also applies to system such as Teak[1].

I have implemented *BBD* in the CLICK backend of the ARCwelder compiler. The compiler compiles dataflow networks that are annotated with functions and types into gate-level netlists. It also generates a global timing engine for that netlist.

2 Asynchronous Dataflow Networks

Let us treat an asynchronous design as a dataflow network.

A dataflow network connects `nodes` with `edges`. Nodes are controllers and edges are *channels*. Nodes communicate with each other by sending packets of data through channels. A channel connects precisely two nodes: a *sender* and a *receiver*. A channel is either *empty* or *full*. If a channel is *empty*, the *sender* can send a packet into it, thereby making the channel *full*. If the channel is *full* , the *receiver* can accept the packet, thereby making the channel *empty* again. When a channel is full, the receiver can inspect the data in the packet on the channel without emptying the channel.

The nodes are typical asynchronous controllers, such as MERGE, JOIN, FORK, etc. Every node connects to one or more channels.

The nodes communicate with each other in a Delay-Insensitive manner[8]. If packets are sent simultaneously through different channels, the design must avoid making assumptions about the relative arrival times of these packets.

2.1 Channel Timing Properties

The ARCwelder compiler maps the dataflow network to a circuit implementation. It maps channels to a data-path, a request wire and an acknowledge wire. The current back-end uses predefined CLICK templates[7] for all the controller components.

The compiler compiles each node in the network separately. In order to facilitate this separate compilation, it allows the node to use certain timing properties of the channels through which the node connects to the rest of the network. These properties are computed by the global timing analysis engine.

An example of a channel property is the *channel response time*. *Channel response time* is a lower bound of the time between the component accessing a channel and the response by the network.

The *channel response time* can be used to guarantee minimum delays along certain paths in the component. This in turn may guarantee the correctness of the component through relative timing without delay insertion[9].

Another channel property is the *ControlLag*. Let in_{ct} be the time at which a control event arrives through channel *in*. Let in_{dt} be the time at which the accompanying data are valid on channel *in*. Then:

$$\mathit{ControlLag}(in) \quad \leq \quad in_{ct} - in_{dt} \tag{1}$$

If the *ControlLag* is positive, the data arrive first. This means that the channel adheres to the traditional bundled data protocol, and the control signal can be used unmodified to capture the data; c.f. Figure 1. If the *ControlLag* is negative, the control could arrive first; c.f. Figure 2. In that case, we need to delay the control signal by at least *ControlLag* to capture the data safely. The *ControlLag* property is an implementation of the *BBD* convention.

The main reason for choosing a bound in *BBD* is that a bound allows us to collapse multiple values into one. If a point in the design has multiple incoming paths, with different *ControlLag* values, there might be multiple different *ControlLag* values on its output channel. A bound allows us to pick a single number. We pick the least number because it is *safe* to compensate for the least number when synchronizing control and data. The least number occurs when the control is most ahead of the data. Thus it requires the largest amount of control delay insertion to synchronize the control signal with the data.

The timing properties of the channels are made available at the channel interface of the component. This allows the component to take its own internal delays into account as well. If it takes, for instance, at least 8 units of time for a storage element to generate the flipflop positive clock edge from the control signal, it can reduce the amount of necessary delay insertion by 8.

Similar to the *ControlLag* property is the *ReleaseLag* property. This property uses *BBD* to bundle the *Acknowledge* signal with the virtual event that permits release of data. This property is necessary to optimize dataflow graphs where one of the Acknowledge signals is a bottleneck on the critical path. Although this is also a relevant property in control overhead optimization, we will not discuss this further in this paper.

3 Computing *ControlLag*

ARCwelder uses a global timing engine to compute the channel properties for the channels in the design. The compiler generates code that delegates computation of the properties to the responsible parties. These responsible parties may in turn request channel timing properties of other channels in the network. In essence the code traverses the dataflow network recursively to compute the properties.

To make this work, every node must be able to compute the *ControlLag* on its output channels based on the *ControlLag* at its input channels. It may use local static timing analysis to compute timing paths in its own implementation as well. For these local timing paths we use an ordinary timing analysis tool. In ARCwelder we use an in-house timing tool `tpshell` that uses unit delays for gate delays, but tools such as PrimeTime[11] or Encounter[3] can perform this analysis as well.

In order for channel property computations to terminate, every possible computation path in the design must eventually reach a controller that produces a constant value for that channel property. The storage component is such a component for the *ControlLag* property. The storage component synchronizes data and control, and issues the new output control event with a constant *ControlLag*. This *ControlLag* is usually a small number, for instance if there is a reset gate on the control path.

If a design fails to provide termination for these properties, there is something wrong with the design. For instance, it may contain a cycle with no storage elements.

Design changes, such as delay insertion, may cause many channels to change their timing properties. It is important that the timing properties at all channels can be recomputed easily. Thus, rather than performing a static timing analysis, the compiler generates (TcL) code that can recompute the timing properties at the channels.

Sometimes it is necessary to insert delays in the design to guarantee proper functionality. These delays may be necessary for proper synchronization, but also to guarantee relative timing constraints, or to guarantee minimal pulse-widths. If the global timing engine is used to compute required delays, then global timing has to be recomputed after delay insertion. It is possible that the new global timing would require new delay insertions. Furthermore, it is possible that the recompute-insert cycle would never terminate. We can formulate constraints in such a way that this cannot happen. A full explanation falls outside the scope of this paper.

The order of delay insertion has an impact on the amount of delay insertion required. This is left for future investigation.

4 Dataflow Example with *BBD*

In this section I show a partial dataflow design. I will use this design to illustrate the power of the *BBD* convention in optimization of the design.

The design in Figure 3 shows a network composed of controllers. Figure 3 also shows the functions and the types that are relevant to this network. The edges are annotated with their associated types.

The controllers are described informally, using a pseudo DI-algebra[5] description, in Figure 4.

The example design shows a filter with two inputs. One input contains a tuple of Words. The other input contains a unary encoded Boolean that will be used (by **D**) to decide which path to take. The upper path adds the two inputs, the lower path just copies the tuple. The merge (**M**) merges the two paths, after which both elements in the tuple are incremented by 1. Finally, the data are stored in the storage element, which is marked with an X. The storage produces both an acknowledge that travels back to the input, and the data to the output.

The first number on every edge is the *ControlLag* that was computed by the tool. We used a timing analysis tool that measures time in integer units.

The negative *ControlLag* at the input of the storage element indicates that if we want to capture the data, the time between the arrival of the control signal and the actual capturing of the data has to be at least 29.

The positive *ControlLag* on the lower input of the MERGE indicates that the data have been valid at least 16 units of time before the control signal arrives at that input.

Note that there are only two places in this design that require full data/control synchronization. The DISTRIBUTOR must have the data on the select channel to send the appropriate output control signal, and hence has to synchronize. It has to synchronize on the select channel only. The STORAGE has to synchronize its input channel. The *ControlLag* values computed by the timing engine prescribe the amount of delay insertion required for these synchronizations.

The MERGE does not necessarily have to synchronize, but the data cannot pass through it until after the proper paths have been set.

5 *BBD* design analysis and optimization

Not only can we use *BBD* to postpone synchronization, and reduce the amount of delay inserted, we can also use it to analyze designs and reduce their control overhead.

Consider the previous example design. Let us assume that the lower path ,through the **skip** join, is the path that we expect to be most used. We want to optimize performance for that path.

5.1 Data delay analysis

Let C be a component with an input channel in and an output channel out. Control signal in_R is the control event on channel in. Suppose that C responds to a transition on in_R, by producing a control transition on out_R.

We define the virtual data-event (x_D), for any channel x, to take place at the time that we can sample the data safely, according to the controllag on the channel. The time of the virtual data event on channel x is given by equation (2).

$$x\_D \quad = \quad x\_R - \mathit{ControlLag}(x) \qquad (2)$$

That is, we replace the $\leq$ in equation 1 with equality, and the actual data event with the virtual data event.

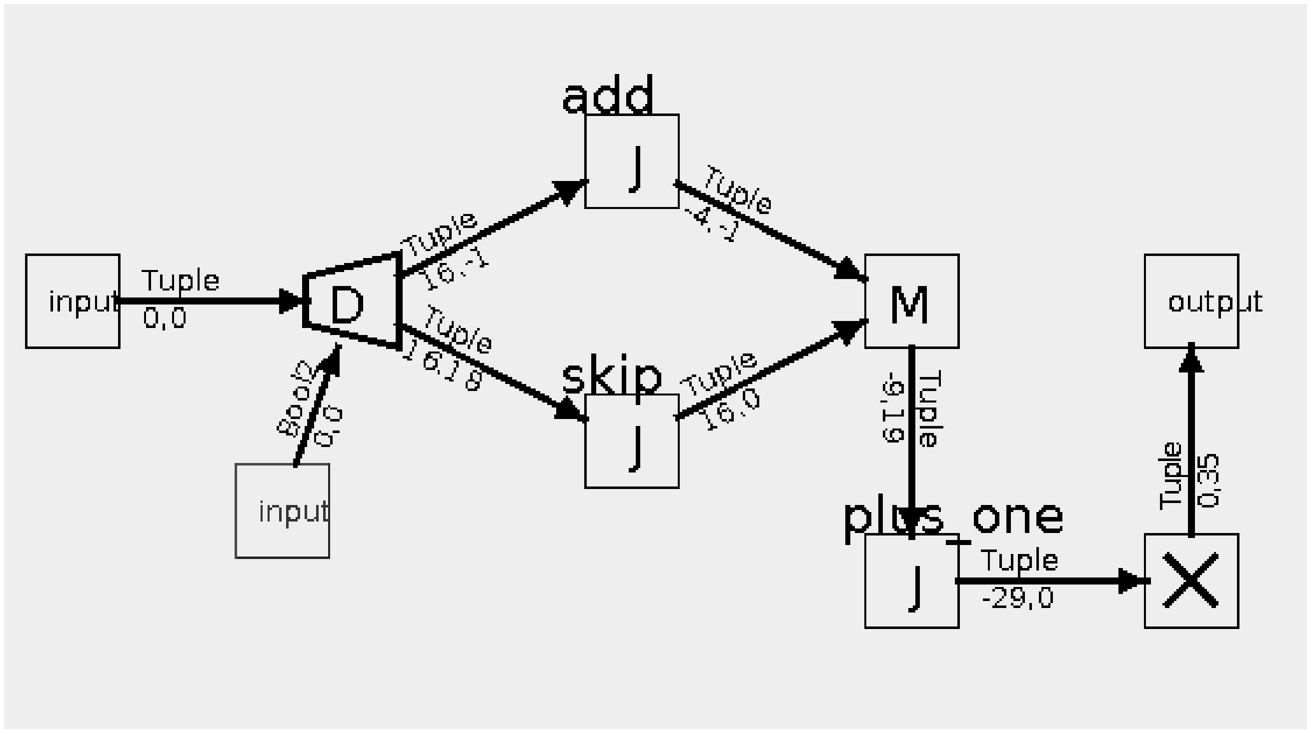
The actual data event may be earlier than the virtual data event. However, if we base our synchronization on controllag values, the virtual data event is the data event that we synchronize with.

If we analyze the waveform of a run of the design, it is easy to find the *ControlDelay* (CD) of a component by finding the time between the in channel request transition and out channel request transition. $out\_R - in\_R = CD$. The control delays along the critical path are given by the second values of the edges in Figure 3. The **skip** join for instance, shows a CD value of 0 on its output channel. This indicates that the control signal is propagated without delay by this component.

From the CD value, and the statically determined ControlLag values, we can derive the virtual data-delay:

$$\begin{aligned}
& out\_D - in\_D \\
= \quad & \{ \text{ term rewriting } \} \\
& ((in\_R - in\_D) - (out\_R - out\_D)) + (out\_R - in\_R) \\
= \quad & \{ \text{ definition (2), definition } CD \ \} \\
& (\mathit{ControlLag}(in) - \mathit{ControlLag}(out)) + CD
\end{aligned}$$

Thus, based on the two *ControlLag* values, and the time between the two Request transitions, we can determine a lower bound on how much the virtual data events are delayed. The **MERGE** component, for instance, shows an input *ControlLag* of 16, an output *ControlLag* of -9 and a CD value of 19 on the path that we want to optimize. It follows that the virtual data-delay through the merge is $16 + 9 + 19 = 44$. Since the virtual data



```
typedef bool[2] Bool2;
typedef bool[32] Word;

typedef struct
{
 Word A;
 Word B;
} Tuple;

function add ( Tuple tup ) : Tuple
{ { tup.A + tup.B , tup.B } }

function skip ( Tuple tup ) : Tuple
{  tup }

function plus_one  (Tuple tup ) : Tuple
{ { tup.A  + 32'h1 , tup.B + 32'h1 } }
```

Figure 3 An example ARCwelder design: Input to the ARCwelder compiler has two aspects. The dataflow graph, that is provided through the GUI, and the descriptions of the types and functions that are provided as a text file. The designer can modify the text-file in an editor of choice and reload it into ARCwelder. In this example, the edges are annotated with the type of the data that they are transporting. The first number on an edge is the *ControlLag*. The second number is the *ControlDelay* on the execution path of interest.

```
STORAGE = in?x ; (in ! || out!x ) ; out ? ; STORAGE

JOIN = in?x; out! F(x); JOIN

DISTRIBUTOR =  in?x;select?s;
                 (if (s[0]) out0!x || if (s[1]) out1!;x)
               ; (if (s[0]) out0? || if (s[1]) out1?)
               ; in!;DISTRIBUTOR

MERGE  = [ in0?x -> M0 , in1?x -> M1]
    M0 = [ in1? -> ERROR, out!x -> out?;in0!;MERGE]
    M1 = [ in0? -> ERROR, out!x -> out?;in1!;MERGE]
```

Figure 4 A pseudo DI-algebra[5] description of the various controllers that we use in the example.
A **STORAGE (X)** element waits until the data are available, then captures the data in a local register, and produces a request signal on its output channel and an acknowledge signal on its input channel. The behavior of the storage is basically that of the simple pipeline stage [10]. The storage element is depicted as a box with an X in it.
A **JOIN (J)** has one input channel and one output channel. Associated with a join is a function F, described in the functional language of ARCwelder. In the tool, the Join is generalized to multiple inputs, and synchronizes on the inputs. Hence the name Join. In the design example that we present later, we need only single-input Joins. The function that is associated with the Join must be expressed in the ARCwelder input text. Figure 3 shows a number of function and type definitions.
A **DISTRIBUTOR (D)** has two input channels. The `select` input channel receives a unary encoding of the output channels to which the `in` channel must be copied. The distributor that we use in our example has 2 output channels, and hence a 2-bit wide select channel.
A **MERGE (M)** waits for one of its input, which it will copy to the output. It is illegal to send concurrent multiple inputs to a merge. Readers that are familiar with the DI algebra may observe that it is legal to send the next input after an output has been produced, and before the other input is acknowledged. This is on purpose.

events are the events that we use to synchronize data and control, this virtual data-delay is the effective data-delay on the path.

If we combine the data-delay value with our knowledge of the components, we can find possible optimization areas. Consider again the example in figure 3.

The first number on the edges is the *ControlLag* on those edges. The second number is the control-delay when exercising the most frequent path. I marked the upper path control delays surrounding the 'add' Join by -1 to indicate that we are not using them. Control delays have to be positive.

For every node, we have a minimal data-delay, which is the logic on the data-path. For instance, a merge has a multiplexor on its data-path, and a join has computation logic on its data-path.

By analyzing the data-delay on the different nodes, we can create the table in Figure 5 for the critical path.

We find that most of the control overhead is located in the Merge(**M**). There is an operational explanation for this. The Merge that we use is quite complex, and requires a long time to set up the data path. In this case, the positive ControlLag tells us that the control signal arrives **after** the data. This means that the setup of the data path will occur after the data arrived, and this adds an additional data delay.

node	in cl	out cl	CD	min delay	act delay	over head
D	0	16	18	0	2	2
SKIP	16	16	0	0	0	0
M	16	-9	19	4	44	40
PLUS	-9	-29	0	20	20	0
STORE	-29	0	35	4	6	2

Figure 5 Timing properties: cl is *ControlLag*(), *CD* is control delay

5.2 Abstraction

The numbers that we need to compute the control overhead were ControlLag (that we could determine stati-

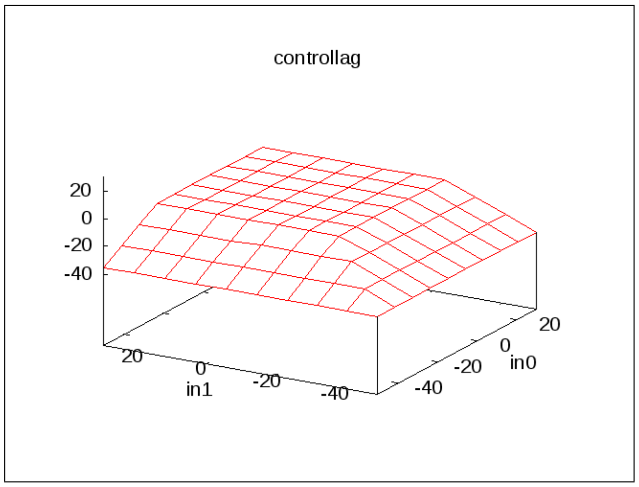


Figure 6 Plot of 2-input merge ControlLag. Recall that negative ControlLag means that control arrives first. The plot shows that when the control is behind, or not far enough ahead, the controllag of the output channel is constant. The reason is that the control signal is now critical on the datapath. Since the control signal is also critical on the control path, the relation between the two on the output channel is constant.

cally), and control delay, which is a value that changes at run-time.

However, if we know which of the input control events is causing the output control event, we can statically determine the time between the input control event and the output control event, and, using the method of the previous section, we can compute the data-delay.

More concretely, let $cl(in0, in1)$ be the statically determined function that computes the *ControlLag*() on the output of a two-input component; c.f. Figure 6. Let (*ControlDelay*)(CD) be the time it takes between the control signal from $in0$ and the output control signal, if $in0$ is the direct cause of the output control signal.

Then, if we know that $in0$ is the channel on the critical path, we can determine $ControlLag(in0) - cl(in0, in1) + CD$. For the merge, this results in the function in Figure 7

Based on the considerations above, we can make the following abstraction: for every controller, we create a function that computes the output ControlLag based on the input ControlLag. Figure 6 shows the graph for the merge element.

For every controller and every input channel, we can create data-delay functions. These data-delay functions are valid only if the input channel for which they were created is the one that actually caused the output event. In Figure 7 we show the datadelay of the merge,

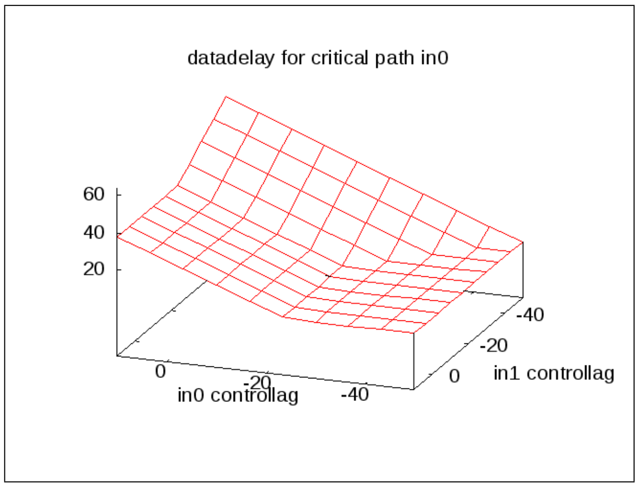


Figure 7 Plot of 2-input merge data-delay when in0 is critical. Recall that negative ControlLag means that control arrives first. It shows that the points of least data-delay are found when the controllag on in1 is not much more than the controllag on in0 and the controllag on in0 should be at most -20.

in our design, where the lower path input (in0) is the one producing the data.

The ControlLag plots can be determined statically, and in a first order approach do not even require knowledge of the layout of the design.

5.3 Optimizing the example

We have already seen that the merge was causing serious control overhead. According to Figure 7, reducing the *ControlLag* on its input channels will decrease control overhead. By moving the **plus_one** join across the merge and duplicating it, we create the improved design of Figure 8. We reduce the critical input ControlLag to the merge, and this reduces control overhead.

Figure 9 shows the control overhead table for the new design.

We find that the control overhead on the Merge, and on the whole path has reduced significantly. The MERGE could benefit from an even smaller *ControlLag* on its critical input, but this may not be achievable in this design.

Based on the two functions in our *ControlLag* abstraction, we are able to quickly optimize dataflow graphs to reduce data-delay. This is the key to control overhead elimination. For now, we are doing this by hand. In the future, we may add automatic optimization strategies.

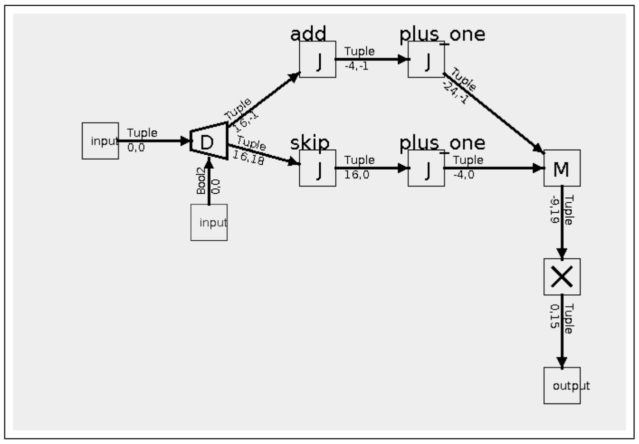


Figure 8 Optimized design example. The **plus_one** Join has moved over the merge to the two input paths of the merge

node	in cl	out cl	CD	min delay	act delay	over head
D	0	16	18	0	2	2
SKIP	16	16	0	0	0	0
PLUS _ONE	16	-4	0	20	20	0
M	-4	-9	19	4	24	20
STORE	-9	0	15	4	6	2

Figure 9 Optimized timing properties

6 Conclusions and possible future work

We have presented the *Bounded Bundled Data* protocol, and have shown how it can be implemented in a global timing analysis tool.

We showed how *BBD* can be used to abstract timing properties from a design in such a way as to allow re-organization of a design to reduce control overhead.

The current implementation of *BBD* is already a large improvement over the algorithm of [7]. Both in its structural approach as well as its impact.

It may be possible to rewrite a dataflow graph automatically to reduce control overhead. We have not investigated algorithms to do so at this point, but the characterizations in this paper are abstract enough to implement and experiment with such algorithms without having to understand the entire compiler chain. It is very possible that the control overhead optimization problem may be solved by existing network optimization algorithms.

We were able to create a small micro processor template, with branching and merging computation paths, where the control overhead was reduced to a single digit percentage of the performance[6]. This was done manually by analyzing the graph in a way similar to that presented in the example above.

There are many feedback and optimization mechanisms that the ControlLag transform graphs can support. We still have to figure out which of those are most beneficial to the designer.

The analysis also suggests that certain new components may be beneficial. In particular a storage component that produces an early request, or an early acknowledge signal. Such components cannot provide a foundation for the analysis as a regular storage component can, but may be inserted at key points to reduce data-delay.

The order of delay insertion that is generated by the constraint-repair functions in the back-end can impact performance. It is conceivable that a particular order of insertions would guarantee an optimal result.

Acknowledgments:

This work was sponsored by a US government grant and by industrial donations.

References

[1] Andrew Bardsley, Luis Tarazona, and Doug Edwards. Teak: A token-flow implementation for the balsa language. In *In Proc. 9th International Conference on the Application of Concurrency to System Design*, 2009.

[2] Peter A. Beerel, Mike Davies, Andrew Lines, and Nam-Hoon Kim. Slack matching asynchronous designs. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 184–194, March 2006.

[3] Cadence homepage. `http://www.cadence.com`.

[4] Gennette Gill and Montek Singh. Bottleneck analysis and alleviation in pipelined systems: A fast hierarchical approach. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 195–205, May 2009.

[5] M. B. Josephs and J. T. Udding. An overview of DI algebra. In T. N. Mudge, V. Milutinovic, and L. Hunter, editors, *Proc. Hawaii International Conf. System Sciences*, volume I, pages 329–338. IEEE Computer Society Press, January 1993.

[6] (censored for anonymity).

[7] Ad Peeters, Frank te Beest, Mark de Wit, and Willem Mallon. Click elements: An implementation style for data-driven compilation. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 3–14, May 2010.

[8] Jens Sparsø and Steve Furber, editors. *Principles of Asynchronous Circuit Design: A Systems Perspective.* Kluwer Academic Publishers, 2001.

[9] Kenneth S. Stevens, Ran Ginosar, and Shai Rotem. Relative timing. *IEEE Transactions on VLSI Systems*, 11(1):129–140, February 2003.

[10] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[11] Synopsys homepage. `http://www.synopsys.com`.