FIFO Sizing for High-Performance Pipelines

Cristian Soviani and Stephen A. Edwards
Columbia University, New York
{soviani, sedwards}@cs.columbia.edu

Abstract

Performance-critical pipelines—such as a packet processing pipeline in a network device—are built from a sequence of simple processing modules, connected by FIFOs. Due to their complex sequential behavior, the worst case throughput, as well as the size of the interconnecting FIFOs, are currently designed using very rough heuristics. Such systems are usually validated by simulation, or worse, field testing.

In this paper, we propose a methodology that address these two issues. First, we propose a fast technique for computing the maximum possible throughput assuming unbounded FIFOs. Then, we describe two algorithms, one exact, one heuristic, that compute minimum FIFO sizes that can achieve this throughput (i.e., FIFOs that do not introduce bottlenecks).

Experimental results suggest our algorithm is applicable to pipelines of at least five modules with runtimes generally in minutes. Since such a computation is only needed a few times for any design, we consider our technique practical.

1 Introduction

High performance pipelines are critical in many modern digital systems. A typical application is a packet processing pipeline found on the line cards of modern network switches.

The performance requirements of such applications (perhaps tens of gigabits per second) mean that the function of individual pipeline modules tend to be very simple; their complexity arises when they are chained together.

In contrast with a pipeline in which each stage does a little bit more of a larger, regular computation (e.g., multiplication), the computation of each module in the pipelines we consider can vary substantially. For example, one module might perform a VLAN (Virtual Local Area Network) computation; the next may handle PPPoE (Point-to-Point Protocol over Ethernet). These very common protocols use Ethernet packets with augmented headers to provide additional functionality.

Another particularity of the pipelines we are considering is that the data flow through each module is not constant: some modules may insert or remove data. Specifically, each module has state and is free to perform different computations at different times depending both on its state and the data fed to it. As such it is not always ready to produce or receive more data. Its environment must comply with such wishes.

Such pipelines consist of modules connected by FIFOs to mitigate the effect of variable data rates. Such FIFOs decouple the behavior of the modules, making the pipeline's overall throughput follow the average module throughput.

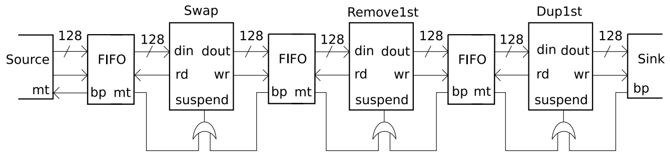

Figure 1: A packet pipeline: how big should the FIFOs be?

Consider Figure 1, a pipeline from a network switch. We want to process packets that start with an arbitrary number of 112-bit Ethernet headers followed by a payload.

The first module swaps the first two headers, the second removes the first header, and the third duplicates the first header. Each module performs its operation on certain packets (those with a matching "type" field) and leaves the others alone.

This pipeline can modify packets in eight different ways. Although this simple pipeline could be replaced by a single module, stringing these operations is more representative of a real packet processing pipeline.

The first module leaves the packet length unchanged, but the second and third may shrink and increase it respectively: the flow through the pipeline is not constant. Moreover, this flow varies according to the contents of the packets.

The combination of data- and state-dependent computation in the modules plus FIFOs makes this a complex sequential systems. Our goal is to address two related questions.

First, what is a pipeline's worst-case throughput? One rule of thumb is that typical pipelines work at 50% throughput, so doubling the clock frequency should be enough to process any data pattern. Our experiments show that for real pipelines, this approximation is often wasteful and may even be incorrect.

Second, given an achievable throughput, what are the minimum FIFO sizes for which we can guarantee that performance? Here, the designer usually over-approximates these sizes using his experience. However, we notice that individual modules are simple, and consist mostly of combinational logic, while FIFOs contain many sequential elements. Over-provisioned FIFOs waste chip area and power and are therefore obvious candidates for optimization. Our techniques allow us to safely reduce FIFO sizes, i.e., without fear of decreasing the performance of the pipeline.

Even for such a simple pipeline, answering questions about throughput and FIFO size is challenging and not something one can do by hand.

Using the methods in this paper, it takes less than thirty seconds to determine the worst-case throughput of this pipeline is

0.6 and that this can be guaranteed with a minimum total FIFO capacity of fourteen (specifically, 4, 5, and 5 for the three FIFOs in Figure 1). Furthermore, if we decrease the throughput requirement to 0.50, 4,4,4 and 4,5,3 are both valid solutions.

Below, we first define some metrics to characterize the sequential behavior of a single pipeline element (Section 3). Then we analyze the worst-case aggregate throughput of the pipeline assuming ideal FIFO sizes (Section 4.1). Next we propose two algorithms, one exact and one heuristic (Section 5), that can determine these ideal FIFO sizes. We support these results by experiment (Section 6).

2 Related Work

Traditional queuing theory is not able to answer these questions. First, it assumes arrival rates can be modeled stochastically, yet network traffic rarely follows standard distributions such as the Poisson. Second, even if we had appropriate distributions, queuing theory only provides stochastic (perhaps average-case) results. Here we are concerned with the worst case, not a distribution.

Le Boudec and Thiran's network calculus [8] provides a helpful methodology for analyzing the pipeline throughput and FIFO sizing, given modules with variable delays, but it requires that each module has a fixed read and write throughput. This is not the case here, where each module may have highly variable, data-dependent input/output behavior.

Many have addressed analyzing and reducing buffer memory consumption for synchronous dataflow graphs (SDF [9]). While the arbitrary topology of SDF graphs is richer than our linear pipelines, they assume their modules produce and consume data at a fixed rate. As such, SDF admits fixed schedules and very aggressive analysis. For example, Murthy and Bhattacharyya [12] show how input and output buffer space can be shared when exact data lifetimes are known. Such precise analysis is generally impossible for our models since they allow data-dependent production and consumption rates.

Like SDF, latency-insensitive design [3] allows richer topologies than our pipelines, but insists on simple module communication patterns. Lu and Koh [10, 11] attack essentially the same problem as we do, but in this semantically different setting. They propose a mixed integer-linear programming solution. Casu and Macchiarulo [4] is representative of more floorplan-driven concerns: their buffer sizes are driven by expected wire delays, something we do not consider.

The asynchronous community also considers buffer sizing (e.g., Burns and Martin [2]), but again consider richer topologies, simpler communication, and fancier delay models.

The obvious question arises: is worst-case analysis justified for real world applications, or would a statistical one suffice? Are corner cases significant in real data patterns?

The answer is yes. Even though most corner cases will never occur, we have observed that observed data patterns generally do belong to a "corner case." For example, Ethernet linecards are often tested with sequences of all smallest length and all longest length packets, as such traffic often occurs in practice.

3 Single module analysis

Our pipelines consist of a linear array of FSMs (Figure 1). We assume the whole pipeline runs synchronously with a single clock, a typical implementation technique. While we could consider an asynchronous implementation, the analysis of the individual modules would be much more complicated (see, e.g., Burns and Martin [2]). Data for each module comes from the source and goes to the sink through the *din* and *dout* signals. For efficiency, these are wide busses, e.g., 128 bits.

The module interacts with source and sink through two control signals, *rd* and *wr*, which the module asserts when it wants to read and write 128-bit blocks of data. None, either, or both of these signals may be asserted each clock cycle.

Additionally, the source provides the *mt* ("empty") signal, asserted when no data is available (e.g., the FIFO is empty). Likewise, the sink asserts the *bp* ("back pressure") status signal when the sink can not accept data. As shown, these two signals are *OR*ed to form the *suspend* signal, which stalls the module in cycles in which it asserted. Thus, in our model, the module is stalled regardless of whether it intends to read or write in a particular cycle. While it would be possible for the module to ignore the suspend signal in cycles in which it neither intended to read nor write, we expect a module implementing such a rule would have a longer combinational delay that would lead to a lower clock rate and probably a net reduction in throughput. With *suspend* asserted, the module holds its state for that cycle and de-asserts *rd* and *wr*.

To evaluate the performance of a single module, we assume that the source and the sink are ideal, i.e., both *mt* and *bp* are never asserted. In this case, the FSM will never be stalled, but the module may not always assert both *rd* and *wr*. In general, their status may depend on the data being fed to the module.

Let r_i^p, w_i^p, $i = 0, 1, 2, \dots$ be the sequences of values on *rd* and *wr* for a given input pattern p.

We are primarily interested in the worst case throughput.

For the input, the worst case corresponds to sequences r_i^p with the smallest number of 1s in a given time. Formally,

$$R = \lim_{t\to\infty}\left(\min_p \frac{\sum_{i<t} r_i^p}{t}\right). \qquad (1)$$

Intuitively, such a limit exists because we are only considering finite systems that ultimately exhibit repeating behavior. We present a more formal argument later.

R is interesting for pipelines such as ingress packet processors when we want to be able to guarantee we can process any input flow at a given rate. For example, if we find $R = 0.4$, it follows that to guarantee a 200 MS/s input throughput for any data pattern, we have to clock the pipeline at 500 MHz (as $500 \times 0.4 = 200$)

We define W, the worst case output throughput, similarly. This is interesting for pipelines such as egress packet processors, where we wish to guarantee an output rate. In the sequel, we will focus on input flow rates; output flow is symmetrical.

We define RW, the minimum read/write ratio, as

$$RW = \lim_{t\to\infty}\left(\min_p \frac{\sum_{i<t} r_i^p}{\sum_{i<t} w_i^p}\right). \qquad (2)$$

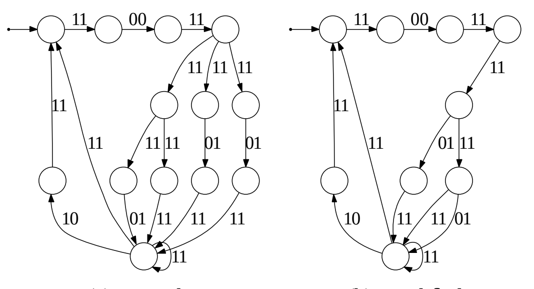

Figure 2: STGs for the DwVLANproc module

3.1 Abstracting the data path

The FSM for a module may have an enormous number of states if it stores an 128-bit sample, rendering a direct application of the above method infeasible. We do the usual trick of dividing the FSM into datapath and control and only consider the STG for the control, which includes the *rd*, *wr*, and *suspend* signals. Any signals from the datapath are treated as independent inputs, meaning there may be extra states and transitions in the abstracted STG. The input/output behavior of the abstracted STG cannot be better than the original, so any performance guarantee we obtain using the abstracted STG also holds for the original system. In the sequel, we only consider simplified STGs.

3.2 Computing R,W, and RW from the STG

We start with the state transition graph of the module's FSM. We build a graph $G = (V,E)$, where V is the set of states, and $E \in V \times V$ is the set of transitions. Any pattern p corresponds to an infinite path in the STG, starting from the reset state. Note that this graph abstracts input and output data; the effect of input data is modeled by multiple outgoing transitions.

We are interested when the *rd* and *wr* signals are asserted, so we assign to each edge $e \in E$ two labels: x^r_e and $x^w_e \in \{0,1\}$.

We present a typical STG in Figure 2a. This can be thought of as an nondeterministic finite automaton with four output symbols that correspond to the four possibilities for reading and writing data. It would be possible to simplify it using a heuristic algorithm to produce the slightly simpler STG in Figure 2b, but we did not implement this optimization.

A simple cycle c is a sequence of edges that forms a non-intersecting path whose tail connects to its head. Let C be the set of all simple cycles in the graph G. We can compute R, W, and RW by considering every simple cycle $c \in C$:

$$R = \min_{c \in C} \frac{\sum_{e \in c} x^r_e}{|c|} \tag{3}$$

$$W = \min_{c \in C} \frac{\sum_{e \in c} x^w_e}{|c|} \tag{4}$$

$$RW = \min_{c \in C} \frac{\sum_{e \in c} x^r_e}{\sum_{e \in c} x^w_e} \tag{5}$$

where $|c|$ is the number of edges in cycle c.

The rationale for these is as follows. R, as defined by (1), corresponds to the pattern with the smallest average read/cycle

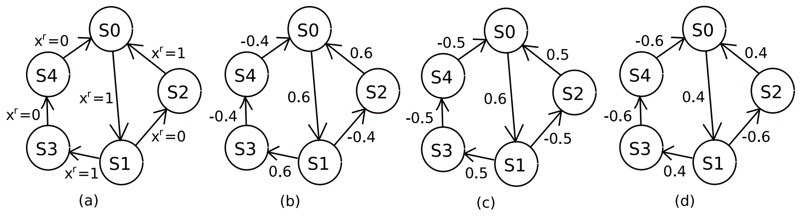

Figure 3: Sample STG to illustrate section 3.3 (a) x^r labels (b), (c), (d) label weights w_e for α = 0.4, 0.5, 0.6 respectively

ratio. This pattern will occur for a periodic behavior of the STG that reads the least on average. Any periodic behavior corresponds to a cycle c in the graph, and the average reading rate of that cycle is $(\sum_{c \in C} x^r_e)/|c|$, exactly (3). A self-intersecting path always has a smaller rate on one of its cycle, so we only consider simple paths. Similar reasoning gives (4) and (5). It also justifies the existence of the limits in (1) and (2).

(3), (4), and (5) compute minimum cycle means [7]. Dasdan [5] uses slightly different notation: we use $\omega(e) = x^r_e, \tau(e) = 1$, $\omega(e) = x^w_e, \tau(e) = 1$, or $\omega(e) = x^r_e, \tau(e) = x^w_e$.

3.3 Using Bellman-Ford

Here, we show how to compute the metrics R and W using a method proposed by Lawner, (see Dasdan [5]). We chose it for its simplicity, given that our STGs are small.

Let $G = (V,E)$ be a directed graph with edge weights w_e for $e \in E$. The $O(VE)$ Bellman-Ford algorithm checks if all cycles in G have positive weight. If they are, it returns the minimum path weight from a given source to each node.

In the sequel, we will ignore any computed path lengths just use Bellman-Ford to check whether $\forall c \in C, \sum_{e \in c} w_e \geq 0$.

Using some simple arithmetic tricks, Bellman-Ford can be used to inexpensively compute certain properties of a graph.

To compute R, note from (3) that

$$R = \max(\alpha) \text{ s.t. } \forall c \in C, \alpha \leq \frac{\sum_{e \in c} x^r_e}{|c|}$$

Assigning $w_e = x^r_e - \alpha$, we have

$$\frac{\sum_{e \in c} x^r_e}{|c|} \geq \alpha \leftrightarrow \sum_{e \in c} x^r_e - |c| \cdot \alpha \geq 0 \leftrightarrow \sum_{e \in c} w_e \geq 0.$$

Thus, we have to find a maximum α such that all cycles are positive. For a given α, we can use Bellman-Ford to check the condition, so we can approximate α arbitrarily well by binary search.

Similarly, to compute RW—see equation (5)—we assign $w_e = x^r_e - \alpha \cdot x^w_e$, which gives

$$\frac{\sum_{e \in c} x^r_e}{\sum_{e \in c} x^w_e} \geq \alpha \leftrightarrow \sum_{e \in c} x^r_e - \alpha \cdot \sum_{e \in c} x^w_e \geq 0 \leftrightarrow \sum_{e \in c} w_e \geq 0.$$

In Figure 3 we illustrate the technique by computing R on a small STG. In Figure 3a, we have the original STG showing the edge labels x^r. By inspection, we find two simple cycles: (S0, S1, S2) and (S0, S1, S3, S4). Their read/time ratios, i.e. $\sum_{e \in c} x^r_e/|c|$, are $2/3 \approx 0.666$ and $2/4 = 0.5$. Thus, according to (3), $R = 0.5$.

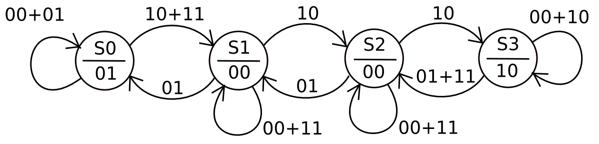

Figure 4: STG of a 3-place FIFO: $I = (wr, rd)$, $O = (bp, mt)$

Since the number of cycles can be exponential in the size of the graph, this straightforward approach is not feasible.

We will approximate R as described above. In Figures 3b, c, and d, we assign $\alpha = 0.4, 0.5$, and 0.6 respectively, and compute the edge weights $w_e = x_e^r - \alpha$, as shown above.

In Figure 3b, both cycles are positive, so the Bellman-Ford algorithm accepts the graph and we conclude $R \geq 0.4$. In Figure 3d, the small cycle is positive but the second is negative, so Bellman-Ford rejects the graph and we conclude $R < 0.6$.

Thus, Bellman-Ford accepts α values greater than the true R and rejects those less than the true R (here, 0.5). We can use binary search to approximate R with arbitrary precision.

The case in Figure 3c is the limit, since here $\alpha = R$. Since this is exactly at the threshold of the all-positive-cycle boundary, it is not surprising that one cycle has exactly zero length.

4 Connecting modules

We build a linear pipeline by chaining various modules and inserting FIFOs between them. Figure 1 illustrates this structure.

We assume that the outputs of a FIFO are driven directly from flip-flops, i.e., that there is no combinational path from input to any output. While this means it always takes at least one clock cycle for a FIFO to react (e.g., by asserting the backpressure signal), such an assumption greatly simplifies the logic synthesis problem by making it easy to optimize the timing of each module and FIFO in isolation. Note that the one-cycle delay means that typically one additional FIFO stage is necessary over what a “faster” FIFO might demand. We also abstract its data path when constructing the STG for a FIFO. Thus for a three-entry FIFO, we obtain the STG in Figure 4.

We wish to know the throughput of the complete pipeline.

First, we will attempt to compute the pipeline’s throughput under the assumption that the FIFOs are “large enough.” Specifically, we want to know the highest throughput possible when the FIFOs are large enough so that they are never a bottleneck. If the FIFOs are infinitely large and assumed to be filled, they never exert backpressure and are always ready to present more data. This is ideal but unrealizable.

At the other extreme, with zero-length FIFOs, data can only progress through the pipeline when every module is ready, a restrictive situation in which unsynchronized module behavior causes a dramatic drop in throughput. Instead, we want the case where each FIFO is just big enough to avoid bottlenecks.

Once we have computed the throughput assuming “large enough” FIFOs, our second challenge is to compute FIFO sizes that exhibit this behavior. We describe this in Section 5.

4.1 *Computing the ideal throughput*

We would like to compute the overall read throughput R_{123} of a pipeline such as that in Figure 1.

```
function Ideal-Throughput
    $r \leftarrow 1$
    for $i \leftarrow n \ldots 1$ do
        $r \leftarrow \min(R_i, r \cdot RW_i)$
    return $r$
```

Figure 5: Computing R assuming optimally-sized FIFOs

To do so, we assume that each FIFO is big enough to compensate for any spurious activity between the modules it connects. We do not consider the FIFOs to be infinite, but instead assume the average data production and consumption rates of the modules in the pipeline are balanced in the limit. I.e., we assume that FIFOs do fill and empty, the overall throughput is such that each FIFO maintains its average fill level.

First, we compute the individual R and RW for each module using the method from Section 3.3. We denote them R_1, R_2, R_3, RW_1, RW_2, and RW_3. Then we consider the pipeline modules from right (output) to left (input).

For module M_3, since it has an ideal sink, R_3 is the throughput M_3 guarantees to accept from upstream; the actual flow entering M_3 in the overall pipeline worst-case may be smaller if the bottleneck lies upstream (i.e., in modules M_1 or M_2).

Looking at M_2, we have to consider two cases. For some data patterns, M_2 may write slower than M_3 can read. In this case, M_2 is the bottleneck, and the FIFO between them will become empty in some clock cycles, causing M_3 to stall. In this case, $R'_{23} = R_2$.

For other patterns, M_2 may want to write faster. In this case, M_3 is the bottleneck, the FIFO will become full in some clock cycles, causing M_2 to stall. For this case, $R''_{23} = RW_2 \cdot R_3$.

Considering both cases, we find $R_{23} = \min(R'_{23}, R''_{23})$.

We similarly move leftwards and compute R_{123}; the complete algorithm is listed in Figure 5.

Computing R_i and RW_i requires building the explicit STG for each module, but as in practice they seldom exceed twenty states; the observed running time is negligible.

This fast algorithm is well-suited to use inside a high-level synthesis design-space exploration loop. Usually, each module admits several implementations that are not sequentially equivalent and have different costs. Moreover, modules can be split or adjacent modules merged, i.e., to vary the pipeline granularity. This algorithm makes it possible to quickly compare variants.

5 Computing FIFO Sizes

In the last section, we showed how to compute the maximum (module-limited) throughput of a linear pipeline assuming ideally sized FIFOs. We now address the problem of finding the smallest FIFOs that can achieve that throughput.

We do this using a guided search that can test whether a particular assignment of FIFO lengths can achieve a particular throughput. The simplest way to determine this (i.e., R for a complete pipeline) is to build the product machine of all its modules and FIFOs and compute R using the algorithm from Section 3.3. Of course, the size of this product machine grows exponentially with the length of the pipeline, quickly making it impractical to implement explicitly. Nevertheless, we were able to use it on small samples to validate the implicit technique we describe below.

5.1 *Verifying throughput using model checking*

Here we use model checking to find out if a given pipeline with FIFOs of given sizes can sustain a certain throughput. Later, we will use this as the core of a search algorithm for determining the minimum-sized FIFOs that support the maximum throughput we computed using the technique presented in the previous section.

Consider a pipeline with n modules $M_i, i = 1, 2, \ldots, n$. At the beginning, we add a source FSM that simply outputs data at a constant rate: the throughput we want to check, T. We denote this additional module M_0.

Because we model the input behavior as a simple periodic (finite) FSM, we can only use values for T which are a ratio of small integers. The interesting cases are when $T = R - \varepsilon$, with $\varepsilon \geq 0$ small, i.e., we try to achieve the throughput we computed assuming sufficiently large FIFOs.

Between the $n+1$ modules we place n FIFOs. We denote the size of the FIFO between M_{i-1} and M_i as $f_i, i = 1, \ldots, n$.

We start with STGs of the pipeline elements M_i and F_i, described by KISS models [6]. We encode each of them using the one-hot algorithm in SIS [13], then assemble the resulting BLIF files, connecting handshaking signals and adding stalling logic. We tie the *bp* input of the last module M_i to 0, i.e., to an ideal sink. The source module M_0 has no *suspend* input, as it produces data at a fixed rate. Instead, our goal is to check that the *bp* output of F_1 will never be asserted, meaning the pipeline can accept the given throughput.

In this point we simply use the *check_invariant* algorithm from the VIS package [1], to verify that, regardless of the current state of the overall system, the property $bp_1 = 0$ holds, i.e., the first FIFO never becomes full and would block output from the first module. This answers the throughput question.

5.2 *FIFO size monotonicity*

Since we use costly model checking to determine whether a pipeline configuration can achieve a given throughput, we make the following observation to reduce the search space.

For two pipelines with the same modules M_i, but with different FIFO sizes f_i and f'_i respectively, we find

$$\forall i, f_i < f'_i \text{ implies } R < R'. \tag{6}$$

This is because increasing the size of a FIFO can never decrease throughput: decreasing throughput requires more backpressure, but a larger FIFO never induces any.

This is not a total ordering, e.g., it does not discriminate when $f_i < f'_i$ and $f_j > f'_j$ for some $i \neq j$. Nevertheless, it helps to decrease the search space when trying to find overall minimum FIFO sizes. When $\forall i, f_i < f'_i$, we write $F \prec F'$.

5.3 *An exact depth-first search algorithm*

Here we present an exact search algorithm for determining minimum FIFO sizes. It is often too slow, so in the next section we accelerate it with a heuristic. Its runtime is sometimes practical; we also use it to evaluate our heuristics.

Figure 6 shows the algorithm, which is a variant of a basic depth-first search. To prune the search space, it uses the FIFO size monotonicity property (Section 5.2), which is checked by the GeThanAny and LeThanAny auxiliary functions.

The core function is ThroughputAchieved, which calls the VIS model checker (Section 5.1), and decides if a given assignment of FIFO sizes can achieve the desired throughput.

The algorithm first considers pipelines with all FIFOs of size one, then all of size two, etc., until a feasible one is found; this is the starting place for the search.

The algorithm maintains three lists of FIFO size assignments: GOOD, BAD, and TRY, which contain the fifo sizes which are proven to be good, bad, and not checked yet. The Succ function returns the next points in the search space to be checked if the current state is good. The depth-first behavior arises by adding and removing elements from the beginning of the TRY list in a stack-like fashion.

The MinSize function returns the best solution found; our cost metric is simply the sum of FIFO sizes, reflecting their area. However, a more complicated metric can be used.

5.4 *A heuristic search algorithm*

We find the exact algorithm in Figure 6 too slow. Instead, we propose the heuristic search algorithm in Figure 7. This does not guarantee an optimal solution, but in practice appears to be able to produce solutions close to it and runs much faster.

Like the exact algorithm, this one starts by considering FIFO sizes of all one, then all two, etc., until a solution is found. Then, it attempts to decrease the largest FIFO. When decreasing the size of this FIFO would violate the throughput constraint, we mark it as “held” and do not attempt to reduce its size further (the H array holds the “held” flags). The algorithm terminates when all FIFOs are marked as “held.”

This algorithm can miss the optimal FIFO size because it assumes the FIFO sizes are independent, which is not true in general. Constraints among FIFO sizes can be fairly complex, for example, increasing the size of one may enable two or more other FIFOs to be reduced (note that this does not violate our monotonicity result of Section 5.2). Nevertheless, we find this heuristic algorithm works well in practice.

To further decrease the running time, we have also explored the case where all FIFOs have the same size, and got slightly worse results compared to the proposed greedy algorithm. In fact, a whole class of heuristic algorithms can be derived from the described method, depending on the cost function to minimize, as well as on the desired trade-off between running time and accuracy.

6 Experimental Results

In our experiments we use four modules (A, B, C, D) taken from a packet processing pipeline in a commercial ADSL-like network linecard. We synthesized them for a 32-bit wide pipeline. In practice, 64- or 128-bit busses are more common, meaning the modules will have fewer states and require smaller FIFOs. We chose complex modules to illustrate our algorithms.

A, B, and C are complex modules that swap, insert, and remove VLAN tags from packets at different points in a pipeline.

```
function GeThanAny($F$, list)
    for $F' \in$ list do
        if $F \succeq F'$ then
            return true
    return false

function LeThanAny($F$, list)
    for $F' \in$ list do
        if $F \preceq F'$ then
            return true
    return false

function MinSize(list)
    s $\leftarrow \infty$
    for $F' \in$ list do
        $s' \leftarrow \sum_i F_i$
        if $s' < s$ then
            $F \leftarrow F'$
            $s \leftarrow s'$

function Succ($F$)
    $S \leftarrow \emptyset$
    for $0 \leq i < n$ do
        $S \leftarrow S \cup (F_0, \ldots, F_{i-1}, F_i - 1, F_{i+1}, \ldots)$
    return S

function SearchMinFIFO
    $s \leftarrow 1$
    $F \leftarrow (1, 1, \ldots, 1)$
    repeat
        BAD.push_front($F$)
        $s \leftarrow s + 1$
        $F \leftarrow (s, s, \ldots, s)$
    until ThroughputAchieved($F$)
    GOOD.push_front($F$)
    for $F' \in \text{Succ}(F)$ do
        TRY.push_front($F'$)
    while TRY.size $> 0$ do
        $F \leftarrow$ TRY.pop_front
        if GeThanAny($F$, GOOD) then
            ok $\leftarrow$ true
        else if LeThanAny($F$, BAD) then
            ok $\leftarrow$ false
        else
            ok $\leftarrow$ ThroughputAchieved($F$)
        if ok then
            GOOD.push_front($F$)
            for $F' \in \text{Succ}(F)$ do
                TRY.push_front($F'$)
        else
            BAD.push_front($F$)
    return MinSize(GOOD)
```

Figure 6: An exact algorithm for computing minimum FIFOs

```
function GreedyMinFIFO
    $s \leftarrow 1$
    repeat
        $s \leftarrow s + 1$
        $F \leftarrow (s, s, \ldots, s)$
    until ThroughputAchieved($F$)
    $H \leftarrow (0, 0, \ldots, 0)$
    while $H \neq (1, 1, \ldots, 1)$ do
        $m \leftarrow -1$
        for $i \leftarrow 1, \ldots, n$ do
            if $H_i = 0$ and ($m = -1$ or $F_i > F_m$) then
                $m \leftarrow i$
        $F_m \leftarrow F_m - 1$
        if not ThroughputAchieved($F$) then
            $H_m \leftarrow 1$
            $F_m \leftarrow F_m + 1$
    return $F$
```

Figure 7: Greedy Search for minimum size FIFOs

A is the most complex module we have found in practice.

Module D is simpler, but not trivial. However, since it does not insert and remove data from the packet, its input/output behavior is trivial. This is the case with many real modules, so we believe including it in the experiments is justified.

To produce different examples, we randomly combined up to five modules (Figure 8). These pipelines may not perform a useful function, but their complexity is representative.

For each sample, the ideal throughput column lists R as computed by the Ideal-Throughput algorithm (Figure 5).

The greedy (Figure 7) and exact (Figure 6) algorithms are run on the slightly smaller throughput listed under "Throughput used." This a small integer ratio (see Section 5.1). The solution, i.e., the configuration with the minimum total FIFO capacity, and the running times are shown in the last four columns.

All the results (throughput, total FIFO size, and running time) vary substantially. In general, shorter pipelines and those containing simpler modules allow for higher throughputs and smaller FIFOs; our algorithms' running times are also faster.

Although the observed running time increases (as one expects) with the pipeline size and the complexity of its modules, we can not ignore some difficult to predict values, such as the long time taken by the "AABCD" module sequence. In fact, the running time taken by the VIS model checker heavily depends on the size of the BDD which implicitly represents the pipeline product machine state space, which indeed is not a trivial function of the module sizes, but, on the contrary, is very sensible to the modules' Boolean properties and their interaction.

The results clearly show module sequence is important, i.e., the interaction of adjacent modules is a critical factor that cannot be ignored for an accurate analysis.

Modules	**Throughput**		**Greedy**		**DFS**	
1 2 3 4 5	**Ideal**	**Used**	**Size**	**Time**	**Size**	**Time**
ABC	0.329	0.250	10	6s	9	18s
CBA	0.337	0.333	11	8s	11	38s
BCD	0.511	0.500	10	2s	10	6s
DCB	0.530	0.500	8	1s	8	2s
ABCB	0.191	0.166	16	1m	13	22m
ABAB	0.123	0.111	16	5m	15	68m
ACCA	0.385	0.333	15	32s	13	8m
CBAC	0.329	0.250	11	11s	11	1m
BBCB	0.167	0.166	17	7m	15	88m
AAAA	0.159	0.142	18	19m	15	326m
ABCDA	0.217	0.200	20	3m	17	150m
DCBAD	0.337	0.333	13	7s	13	1m
AABBC	0.119	0.111	24	409m		
BBCCD	0.288	0.250	17	1m		
CCDDA	0.600	0.500	10	1s	10	1s
DDAAB	0.219	0.200	13	31s	13	5m

Module	**Name**	**States**	**Transitions**	***R***	***RW***
A	VLANedit	24	33	0.600	0.643
B	UpVLANproc	18	40	0.530	0.563
C	DwVLANproc	13	17	0.909	1.000
D	VLANfilter	1	2	1.000	1.000

Figure 8: Experimental results and statistics on 32-bit modules

7 Conclusions

We addressed worst-case performance analysis and FIFO sizing for pipelines of modules with data-dependent throughput, such as those found in network processing devices. We have shown the performance of such a pipeline depends on both its elements and their interaction.

We presented a performance estimation technique for non-interacting modules. It assumes infinite FIFOs and runs fast enough to be used inside a high-level design exploration loop.

Interaction makes the analysis of a real pipeline, with finite FIFOs, difficult. To answer the problem, we propose two algorithms, one exact and one heuristic, which use a model checking algorithm to evaluate the feasibility of candidate solutions.

It can be noticed that the above model checking technique can be applied to systems with arbitrary topologies, in addition to linear pipelines. Unfortunately, this is not possible for the first algorithm, which considers non-interacting modules, but we consider that extending it in this direction might be a very interesting research topic.

The presented algorithms require substantial running time, but we consider them practical since they can be run in parallel with the detailed logic synthesis of the individual modules.

References

[1] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. In *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[2] S. M. Burns and A. J. Martin. Performance analysis and optimization of asynchronous circuits. In *Proceedings of the University of California/Santa Cruz Conference on Advanced Research in VLSI*, pages 71–86. MIT Press, 1991.

[3] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, Sept. 2001.

[4] M. R. Casu and L. Macchiarulo. Throughput-driven floorplanning with wire pipelining. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(5):663–675, May 2005.

[5] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems*, 9(4):385–418, Oct. 2004.

[6] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-4(3):269–285, July 1985.

[7] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309–311, 1978.

[8] J.-Y. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[9] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept. 1987.

[10] R. Lu and C.-K. Koh. Performance optimization of latency insensitive systems through buffer queue sizing of communication channels. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 227–231, 2003.

[11] R. Lu and C.-K. Koh. Performance analysis of latency-insensitive systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(3):469–483, 2006.

[12] P. K. Murthy and S. S. Bhattacharyya. Buffer merging—a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Transactions on Design Automation of Electronic Systems*, 9(2):212–237, Apr. 2004.

[13] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 328–333, Cambridge, Massachusetts, Oct. 1992.