# Fluid Pipelines: Elastic Circuitry meets Out-of-Order Execution

Rafael Trapani Possignolo and Elnaz Ebrahimi and Haven Skinner and Jose Renau
Dept. of Computer Engineering, University of California, Santa Cruz, Santa Cruz, CA, 95064
Email: {rpossign,eebrahim,hskinner,renau}@ucsc.edu

*Abstract*—**Pipeline depth and cycle time are fixed early in the chip design process but their impact can only be assessed when the implementation is mostly done and changing them is impractical. Elastic Systems are latency insensitive systems, and allow changes in the pipeline depth late in the design process with little design effort. Nevertheless, they have significant throughput penalty when new stages are added in the presence of pipeline loops. We propose Fluid Pipelines, an evolution that allows pipeline transformations without a throughput penalty. Formally, we introduce "or-causality" in addition to the already existing "and-causality" in Elastic Systems. It gives more flexibility than previously possible at the cost of having the designer to specify the intended behavior of the circuit. In an Out-of-Order core benchmark, Fluid Pipelines improve the optimal energy-delay point by shifting both performance (by 17%) and energy (by 13%). We envision a scenario where tools would be able to generate different pipeline configurations from the same RTL *e.g.*, low power, high performance.**

## I. Introduction

Cycle time and pipeline depth are set early in design due to their impact on other design parameters. Meeting a desired cycle time requires numerous long iterations between design and implementation. Elastic (or latency insensitive) Systems [1], [2], an alternative to the fixed pipeline paradigm, are based on the assumption that system correctness does not depend on latency (number of clock cycles) between two events, but on their order [3], [4]. This allows for stage insertion later in the design time without breaking circuit correctness [3].

Changing the number of cycles [5], [6] is possible in Elastic Systems but constrained by the presence of sequential loops[1], which significantly reduces its applicability because complex circuits such as processors include loops. In general, an automated flow transforms synchronous circuitry into elastic. Since the flow has no knowledge of the intended behavior of the circuit, it maintains the completion order of events, reducing the circuit throughput [3], [7]. Throughput losses can be mitigated [5] but the whole system remains constrained by the worst sequential loop, even when that loop is not used.

In contrast, Out-of-Order (OoO) execution is omnipresent in modern digital design and improves system throughput. We propose Fluid Pipelines, an evolution of current Elastic Systems that enable unordered completion. Fluid Pipelines rely on designer annotations in the code where ordering can be changed. Fluid Pipelines are a generalization of Elastic Systems, since without user annotations, they behave like Elastic Systems. User defined elasticity has been proposed [8], and is thought to improve design methodologies [4].

[1]Cycles in the graph representing the connections between registers, not to be confused with program loops.

Fluid Pipelines reclaim the throughput losses from the automated conversion [9]. The automated flow of Elastic Systems transforms a sequential circuit to an elastic one by inserting Fork and Join operators. In short, Fork is used when the output of one stage forks to multiple stages, whereas Join is used when parallel data paths reunite, therefore, the inputs of a stage come from separate stages. The Join operator requires all the inputs to be valid in order to proceed, *i.e.*, the inputs to an adder unit need to be ready at the same time for the operation to take place. When there is no dependency between the inputs of a block, a Merge operation is said to take place. Merge differs from Join, because it is triggered when at least one of the inputs is valid (*i.e.*, it has "or-causality"), in addition, only data from one of the inputs is consumed at each cycle. Its dual, Branch, propagates data to only one of multiple output paths, as opposed to sending data to all of them. This behavior is found in many digital designs, like a Floating Point Unit (FPU) with independent operations; or a network router, where packages come from different inputs and propagate to a single output.

We propose a new methodology based on Coloured Petri Nets (CPN) [10], to determine the throughput of Elastic Systems and Fluid Pipelines. The main objective of this methodology is to allow a designer to quickly explore the design space without needing to simulate every design point. Our methodology is faster than RTL simulation for all possible pipeline configurations. For more complex cases, such as a full fledged OoO core, we rely on cycle accurate simulation to determine the system performance.

Our results show that for an OoO core, Fluid Pipelines improve the optimal energy-delay (ED) point by increasing performance by 17% and reducing energy by 13%, when compared to previous Elastic Systems. A simpler FPU benchmark shows even better results, with improvements of up to 176% in performance, and 5% less power consumption. By using CPN models, it is possible to explore the Pareto frontier and select different interesting design points, depending on a specific application.

The contributions of our paper are:

- Fluid Pipelines (Section IV), an Elastic System evolution that improves the Pareto frontier by avoiding the typical throughput loss.
- An evaluation methodology (Section V) using Coloured Petri Nets (CPN) for Elastic Systems and Fluid Pipelines.
- An evaluation (Section VII) of an OoO CPU core and an FPU to quantify the impact of Fluid Pipelines.

## II. Related Work

Out-of-Order and Speculation targeting parallel execution were evaluated in software Dataflow Networks [11]. The authors use the same concepts we propose, but attack the task scheduling problem in OoO cores. The flow speculates which dependencies are true dependencies and trigger re-execution in case of a mis-speculation. Our approach relies on the designer knowledge of the logic to avoid such scenarios.

High Level Synthesis (HLS) [12] allows designers to focus on functionality, while the tools perform pipelining during scheduling [13]. HLS generates traditional synchronous circuits (*i.e.*, not elastic), and thus scheduling is limited by the presence of dependency loops. HLS could leverage Fluid Pipelines to enable recycling in such loops, and are orthogonal in that regard. In fact, this could improve HLS design time by avoiding multiple iterations to meet timing (*i.e.*, by adding flops without going back to the RTL description).

Dimitrakopoulos *et al.* [14] explore the reduction of buffering to support multi-threading in Elastic Systems. Their work presents a certain amount of Out-of-Ordering on an inter thread basis (*i.e.*, no ordering enforced between different threads). Our work allows full Out-of-Order execution. The analogy would be that of a Simultaneous Multithread (SMT) in-order core versus an Out-of-Order core.

Elastic Coarse Grain Reconfigurable Arrays (CGRAs) [15] are an approach for coarse grain reconfigurable logic that relies on elastic interfaces for flow control. Elastic CGRAs use Merge and Branch operators across basic blocks (connecting inputs and outputs from different accelerator units), while Fork and Join are used within basic blocks (in the calculation itself). This is conceptually similar to Fluid Pipelines, but limits where each operator can be used.

To mitigate throughput loss in Elastic Systems, different approaches have been proposed. The Eager Fork operator [2] lets one of the paths start executing even when the parallel path is not ready. Whereas, FIFOs allow for more buffering [4]. Early Evaluation [5] determines which inputs in merging paths are actually needed (such as in a mux), and only waits for those inputs. The next input from other paths is ignored to maintain correctness. Nevertheless, those approaches do not change system semantics. This becomes problematic when one of the paths takes multiple cycles to complete. Then, back pressure propagates to the preceding stages. Fluid Pipelines avoid this scenario by not waiting for parallel paths unless it is needed.

## III. Background

Functionality of an Elastic System depends on the order of its inputs and not their arrival time [16]. Events or *tokens* are meaningful data flowing through a *channel*. A *channel* is a set of wires (*i.e.*, bus) and its associated control signals: *Valid* (V) and *Stop* (S)[2], which determine three states: *transfer* ($V = 1$, $S = 0$), *idle* ($V = 0$) and *retry* ($V = 1$, $S = 1$) [2].

An execution example is shown in Figure 1, where the arrival of a valid token is represented by a number in a given cell. When a result is produced, the token is consumed and

[2]Other equivalent naming conventions have been used, *e.g.*, Elasticity has been expressed in terms of FIFO operation [4].

| Clock Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| A | 0 | 4 | | | | 3 | | |
| B | 1 | | 2 | 3 | | | | |
| A+B | | 1 | | 6 | | | | 6 |

**Fig. 1:** Elastic Systems functionality does not depend on the exact cycle events happen, but rather on their order.

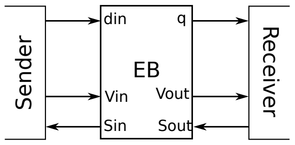


**Fig. 2:** Elastic buffers are the basic construct blocks of Elastic Systems and can be viewed as queues with a limited size.

can no longer be used. Empty cells in the table denote that no new data has arrived in that cycle. Note that the latency between events is arbitrary.

*Elastic Buffers* (EBs) are storage units that replace registers; They include handshake signals both on the input and output interface. Figure 2 shows the interface of an EB with input and output control signals.

### A. ReCycling and Retiming in Elastic Systems

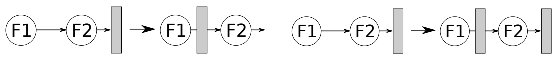
To improve the frequency of Elastic Systems, it is possible to move EBs across circuit blocks (Retiming) [5] (Figure 3a) or to insert additional stages in slower paths (ReCycling) [5] (Figure 3b). Retiming preserves the sequential behavior of the circuit [5] and thus it can be applied mostly without penalties.

In the case of ReCycling, the throughput of a system is limited to the sequential loop with the lowest throughput, calculated as the number of tokens in the loop divided by the number of EBs in the loop [7]. The throughput of a cycle can increase with Early Evaluation depending on how often each event occurs [7], but due to back pressure, there is still a limit on such mitigation. ReCycling is able to reduce cycle time, [17] but may decrease throughput in the case of stage insertion in sequential loops [3], [5], [7].

## IV. Fluid Pipelines

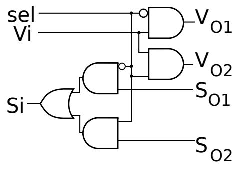
Fluid Pipelines evolve the traditional Elastic Systems to allow breaking the relative completion order. This is accomplished by using four types of operators: Branch, Merge, Fork and Join operators (Figure 4). In Figure 4a, *sel* is a data-dependent selection signal that indicates to which output the data will propagate. The operators can be easily extended to more than two inputs/outputs.

Branch is used when the datapath forks into multiple paths, but data should propagate to only one of them. This is controlled by the selection signal. For instance, an operation in an FPU only needs to propagate to the appropriate functional unit, and the selection signal is encoded by the operation bits.

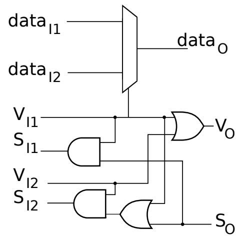


**(a)** Retiming **(b)** ReCycling

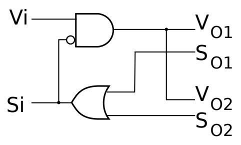
**Fig. 3:** Retiming and ReCycling are used to improve the circuit frequency, but recycling decrease the throughput of Elastic Systems when applied to sequential loops.



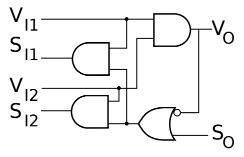
**(a)** Branch



**(b)** Merge



**(c)** Fork



**(d)** Join

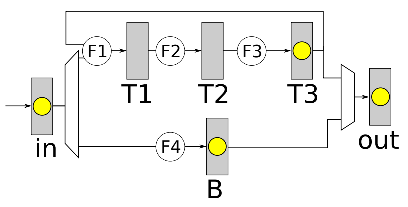
**Fig. 4:** Fluid Pipelines use different operators to translate the intended functionality of a circuit and enable better design space exploration. Branch and Merge are used when the relative order of operations can be broken, while Forks and Joins enforce ordering. Note the difference in the handling of "valid" and "stop" signals.

Merge operates as an arbiter: multiple senders compete for a single channel. The sender that wins the arbitration propagates its data. In our FPU example, a Merge would be used at the output end of the functional units when results from each unit are collected. Another way to think of the Merge is that it fires when at least one of its inputs contains valid data. This is known as *disjoint or-causality* and introduces the *or-firing* rule to the context of Fluid Pipelines. For simplicity and without loss of generality, the proposed implementation in Figure 4b has simple fixed-priority, but can be replaced with any of the existing elaborated arbitration schemes such as Round-Robin.

Merge and Branch cannot be automatically inserted like Fork and Join, because they alter the relative order between events. As a result, the programmer is responsible for inserting them when needed. For example, in a complex Floating Point Unit, just one Merge and Branch pair is needed after the normalization and denormalization stages to indicate that the floating operations can complete out of order. On the other hand, the Fork and Join operators can be automatically inserted in a similar way as the insertions performed in traditional Elastic Systems. Merge and Branch can be performed with direct Verilog/VHDL instantiation or just code annotations. In this paper, we used direct Verilog annotations, and a more automatic solution is left for our future work.

To explain how Fluid Pipelines work, let us analyze the sample execution in the example in Figure 5, where circles represent combinational logic, boxes represent EBs, and the dots inside boxes represent the presence of valid data (tokens). The paths are mutually exclusive (each operation either takes the top or the bottom path), and the mux near the output EB chooses the appropriate path. The instructions can take either the bottom path or the top path in Figure 5b. The execution traces for traditional Elastic Systems and Fluid Pipelines are shown in Table I.

The execution order of Fluid Pipelines is altered (Table I), note how in cycle 3, it is possible to move I3 to the bottom



**(a)** Loop Case

| ID | Path |
|---|---|
| I1 | Bottom |
| I2 | Top |
| I3 | Bottom |
| I4 | Bottom |
| I5 | Top |
| I6 | Bottom |
| I7 | Bottom |

**(b)** Instructions

**Fig. 5:** Toy case to illustrate the Elastic vs. Fluid approaches. Combinational logic is omitted, and Early Evaluation is assumed for elastic. Dots represent registers with a token.

**TABLE I:** Sample trace for the toy case, Fluid Pipelines improve throughput compared to Elastic Systems.

| | Elastic | | | | | | Fluid | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Cycle | in | T1 | T2 | T3 | B | out | in | T1 | T2 | T3 | B | out |
| 0 | I1 | | | | | | I1 | | | | | |
| 1 | I2 | | | | I1 | | I2 | | | | I1 | |
| 2 | I3 | I2 | | | | I1 | I3 | I2 | | | | I1 |
| 3 | I3 | | I2 | | | | I4 | | I2 | | I3 | |
| 4 | I3 | | | I2 | | | I5 | | | I2 | I4 | I3 |
| 5 | I4 | | | | I3 | I2 | I6 | I5 | | | | I2 |
| 6 | I5 | | | | I4 | I3 | I6 | | I5 | | I6 | I4 |
| 7 | I6 | I5 | | | | I4 | I7 | | | I5 | I7 | I6 |
| 8 | I6 | | I5 | | | | | | | | | I5 |
| 9 | I6 | | | I5 | | | | | | | | I7 |
| 10 | I7 | | | | I6 | I5 | | | | | | |
| 11 | | | | | I7 | I6 | | | | | | |
| 12 | | | | | | I7 | | | | | | |

path, while the top path is still executing. This re-ordering is a result of the "or-firing" rule and it is done because it was specified by the user, and not changed by the tool. In a processor core, the reordering buffer performs this function, while in network-on-chips, the reordering is usually not performed. Since this requirement is application specific, it is left out of this manuscript. We assume that any reordering needed is performed in the design. In the case where order should be maintained, regular Fork and Join operators must be used, causing the design to behave similarly to a Elastic System.

### *A. Design Overhead*

In this section, we elaborate how finding points where Merge and Branch operators can be inserted is a simple task because most existing designs are inherently elastic.

Elasticity is omnipresent in digital design. To quantify it, we take a look at various designs in OpenCores [18], an opensource database of digital designs. We counted the number of design implementations that are equivalent (same or inverted signals), partially equivalent (only using one signal or using signals with different meanings), or nonequivalent (not implementing any handshaking) to our handshaking mechanism. We only considered projects marked as "DONE", in Verilog or VHDL and for which the code is publicly available. Out of 270 projects, 35% are equivalent in most blocks, 10% are equivalent in a few blocks, 20% are partially equivalent (in general, only "start" and "done" signals). 25% implement no or an incompatible handshake. The remaining

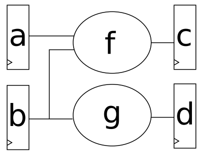


**Fig. 6:** Fluid Pipelines design uses a few design practices to avoid deadlocks. Those are restriction on how to implement a given design and not on which designs can be implemented.

10% are IO operations (debouncer, LED control, . . . ) or only combinational logic (lookup tables, arithmetic operation, . . . ).

These statistics show that the type of handshaking required by Fluid Pipelines is already implemented in most designs, and therefore, Fluid Pipelines will not introduce design overhead. The designer simply needs to annotate the code.

## *B. Fluid Pipelines Deadlock Avoidance*

In Elastic Systems, deadlocks come from extraneous dependencies [4], *i.e.*, one output of a module waits for an input that it does not depend upon to fire. Another issue is the creation of a token in the output before the consumption of one in the input. This is specially a problem in Fluid Pipelines since the designer has more freedom than in previous approaches. This is easily avoided by adhering to the following design practices:

- **No extraneous dependencies:** If an output $o$ of a module does not depend on an input $i$ of that module, then $o$ should be produced regardless of the existence of $i$. Also, the dependency list of $o$ should be a subset of the inputs of the module.
- **Self-cleaning:** A circuit is self-cleaning if whenever it has produced $n$ tokens in its outputs, it has also consumed $n$ tokens from its inputs.

These directives do not restrict which designs are possible, but rather how to implement each design. To make it clearer, let us consider the example in Figure 6. The synchronous module described in the figure has a pair of inputs ($a$ and $b$) and outputs ($c$ and $d$), the value of $c$ depends on the values of $a$ and $b$, while the value of $d$ depends only on the value of $b$. Now, assume a designer wants to implement that module using Fluid Pipelines. There are multiple options available.

The most straightforward implementation of the block would follow the behavior described in Figure 7, which waits until all inputs have valid data, and all outputs can accept new data to perform the operation. This is a violation to the no extraneous dependencies directive and can cause deadlocks depending on the context in which the block is used. For instance, in cases where the output $d$ is connected as a feedback path to $a$, $d$ will only produce output when both $a$ and $b$ are available.

A simple solution to this case is the use of a Fork operator. The Fork operator isolates the handshake handling, and thus avoids the deadlock situation by avoiding the unnecessary wait on a valid signal in $a$ to propagate $d$. An implementation using Fork is shown in Figure 8.

The Self-Cleaning property is needed to avoid buffer overflow. In a circuit that produces $n$ inputs per token consumed where the output of is connected back to its input. For a buffer with size $m$, it is clear that after $m/n$ cycles, the buffer will be full, causing a deadlock.

```
always @ (posedge clk)
  if (a_valid && b_valid)
    if(!c_stop && !d_stop)
      c <= f(a,b);
      d <= g(b);
      c_valid <= true;
      d_valid <= true;
      a_stop <= false;
      b_stop <= false;
```

**Fig. 7:** A straightforward implementation of a circuit may be deadlock prone.

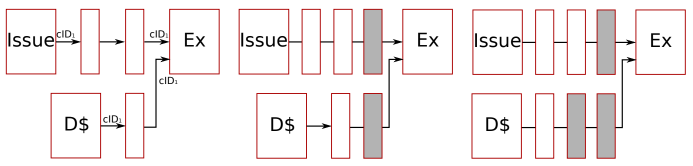
```
module fork(in, out1, out2)
  if(in_valid && !out1_stop && !out2_stop)
    out1 <= in
    out2 <= in
    in_stop <= false
    out1_valid <= true
    out2_valid <= true

module f_and_g(a, b, c, d)
  fork(b, b1, b2);

  always @ (posedge clk)
    if (a_valid && b1_valid && !c_stop)
      c <= b1;
      c_valid <= true;
      b1_stop <= false;

    if (b2_valid && !d_stop)
      d <= b2;
      d_valid <= true;
      b2_stop <= false;
```

**Fig. 8:** Extraneous dependencies can be avoided by using fork to broadcast signals and isolating false dependencies between stages.

## *C. Fluid Pipelines Channel Grouping*

Modern OoO cores rely on the knowledge of the relative number of cycles between two events. For instance, instruction wake up with cache hit speculation relies on the knowledge of how many cycles it will take the data from a cache hit to arrive at the execution unit. Then, the instruction is woken up in time to reach the execution at the same time, avoiding penalties. This information needs to be taken into account when ReCycling is applied, *i.e.*, the number of cycles added or removed from both paths needs to be the same.

To support this behavior, Fluid Pipelines allow the designer to assign IDs to a channel (set of data with associated handshake signals). Channels with the same ID are ReCycled in the same manner, *i.e.*, the same number of stages need to be added/removed from each of the channels with the same ID. There is no requirement that channels share wires or handshake signal and the number of buffers already present in different channels do not need to match. For instance, Figure 9 shows the instruction wake-up and data cache of an OoO core, the channels connecting wake-up to execute and data cache to execute are assigned the same ID, and thus need to be ReCycled by the same amount. A possible ReCyling is shown in Figure 9b, where one extra stage is added (shaded). The circuit in Figure 9c is not a valid ReCycling, since different number of stages is added in each channel.



**(a)** Original **(b)** Valid **(c)** Invalid

**Fig. 9:** Channel IDs allow the designer to constraint what pipeline configurations are allowed to guarantee the functional behavior of the circuit.

## V. New Evaluation Methodology

To find the optimal pipeline depth, a designer or tool must estimate the throughput of a pipeline configuration (*i.e.*, number and position of pipeline stages). For throughput estimation, we propose the use of Coloured Petri Nets (CPN) [10], which allows a fast design space exploration, reducing the need of RTL simulation for every pipeline configuration. CPNs are used for evaluations based on events/transitions and data/tokens which is what we need to perform Branch-like operations.

Petri Nets are defined as a bipartite graph of *places* and *transitions*, connected by *arcs*. Places can contain *tokens* that have data value attached to them (*colour*). The state of the net (the *marking*) is defined by the number and colour of tokens in each place. The initial marking is changed when transitions *fire*. When a transition fires, tokens are subtracted from its input places and added to its output places according to *arc expressions*. There is a *capacity* associated with each place representing the maximum number of tokens in that place, and prevents input transitions from firing.

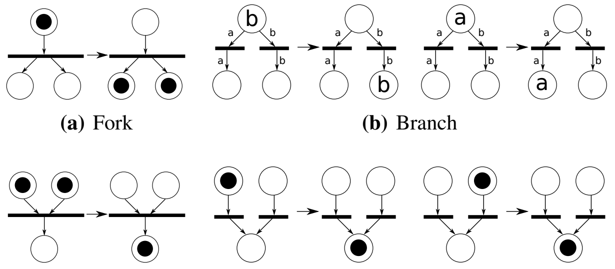
*Definition 1:* A **Coloured-Petri Net** is a tuple CPN $= \langle P, T, A, \Sigma, C, G, E, I, Cap \rangle$:

- P is a finite set of *places*.
- T is a finite set of *transitions*, such that $P \cap T = \varnothing$.
- $A \subseteq (T \times P) \cup (P \times T)$ is a set of directed *arcs*. Let $a.p$ and $a.t$ denote the place and transition connected by $a$ respectively.
- $\Sigma$ is a finite set of non-empty *colour sets*.
- $C : P \rightarrow \Sigma$ is a *colour set function* which assigns a colour set to each function.
- $G$ is a *guard function* that assigns to each transition $t \in T$ a guard function $G(t) : (\varnothing \cup \Sigma)^{|\bullet t|} \rightarrow \{0, 1\}$, where $\bullet t = \{p | (p, t) \in A\}$.
- $E$ is an *arc expression function* that assigns to each arc $a \in A$ an expression $E(a)$, such that the type of $E(a)$ should match $C(a.p)$.
- $I$ is an *initialization function* that assigns to each place $p \in P$ an initialization expression $I(p)$, $I(p)$ must evaluate to $C(p)$.
- $Cap : P \rightarrow \mathbb{I}$ is a capacity function that attributes a maximum capacity to each place.

**Firing Semantics:** Let $M$, a *marking* function, map each place $p \in P$ into a set of tokens $M(p) \in C(p)$. Let $G(t)(M)$ (resp. $E(a)(M)$) denote the evaluation of $G(t)$ (resp. $E(a)$) with the marking $M$. A transition $t$ is enabled, and said to *fire* when $G(t)(M) = true$ and $\forall a \in \{b | b = (p, t), p \in P, b \in A\}, E(a)(M) <= M(a.p)$, and $\forall p \in t\bullet, M(p) < Cap(p)$, where $t\bullet = \{p | (t, p) \in A\}$. The firing updates the marking function to $M'(p) = (M(p)\ E(p,t) \cup E(t,p) \forall p \in P$.

**Timing:** In order to evaluate digital circuits, we need to account for timing, which is not included in CPN models. In regular CPNs, only one transaction fires at a given cycle. Without changing the underlying semantics of CPNs, we modify the model so that *every* transition that is enabled at the beginning of the cycle fires. This is a more accurate description of digital circuits and will help determine the number of clock cycles it takes to execute.

We add one restriction to this formulation. The cardinality of each expression must be 1; this means that for each arc, only one token can be consumed/generated. Also, note that guard functions can only depend on the incoming arcs to a transition. This complies with the constraints defined previously, and thus, avoids deadlocks. The restriction on the cardinality of expressions changes the formalism of CPNs, and a formal analysis of the impact of it is out of the scope of this paper and needs to be further explored in future work.



**(a)** Fork **(b)** Branch

**(c)** Join **(d)** Merge

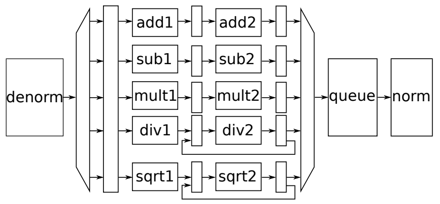
**Fig. 10:** CPN models can be used to estimate the overall throughput of Fluid Pipelines and Elastic Systems.

Figure 10 depicts how the Fluid Pipelines' operators are modeled as CPN transitions. Circles represent places, bars represent transitions, and dots represent tokens in transitions that are not colour dependent while letters represent coloured tokens. Merge operators do not define priority, and thus, conceptually both transitions can occur at the same time, which is compatible with the theoretical formulation of Fluid Pipelines. While places correspond to elastic buffers, transitions do not have a direct translation from the circuit model. However, they can be mapped from the logic.

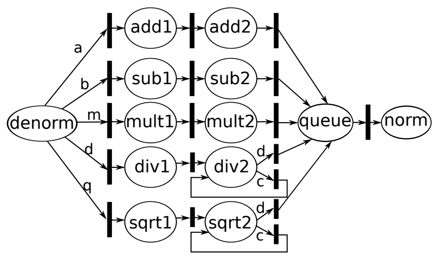
## VI. Evaluation Setup

To evaluate Fluid Pipelines, we consider a fully compliant IEEE-754 in-house FP Unit and a 2-way Out-of-Order Fab-Scalar core [19] designed both as synchronous (for previous approaches), and annotated with Fluid Pipelines' operators.

A functional block diagram of the FPU unit is presented in Figure 11a, and the CPN model used for the performance evaluation is shown in Figure 11b, considering Fluid Pipelines. In this case, the Branch and Merge operators are used. Note how the division and square root modules use the Merge to choose between the loop when the operation is computing or sending the result to the queue when done. Both division and square root take 64 cycles to complete. For regular elastic, the Fork and Join operators are used instead.

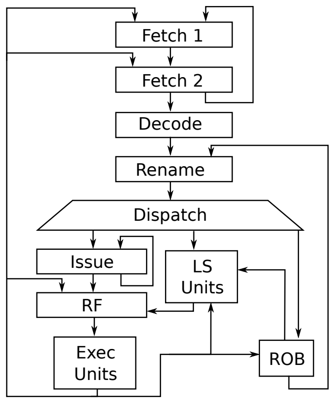


**(a)** FPU



**(b)** CPN Model

**Fig. 11:** CPN modeling can be used to evaluate system performance.



**(a)** FabScalar Core

**Fig. 12:** An OoO core contains a complex structure of nested loops and interactions between blocks. It is used to show the scalability of Fluid Pipelines.

The FabScalar-2W OoO core (Figure 12a) contains nested loops and interactions between blocks and allows us to explore the scalability of the different approaches. Branch operators are used in the dispatch unit, Exec units and issue logic. Merge operators are used after the exec units, in the free register pool handling (ROB to Rename path), and in the next program counter calculation (Fetch 1).

Fluid Pipelines are compared against SELF [5], [14] and LI-BDNs [4]. To implement Elastic Systems, we use an EB implementation with storage capacity of 2. For LI-BDNs, we use queues of size 8. In the SELF implementation adding pipeline stages to all the paths that are parallel to the critical path will yield best performance and that is the performance we have considered in our evaluation.

## *A. Benchmarks*

For the FPU design, we report maximum and average throughput. Maximum throughput is calculated by the using a synthetic workload that only considers the best path (add, subtract and multiply in this case). The average case is calculated as the throughput over a million random instructions.

For the OoO core, only average case is measured. We run all the SPEC2006 benchmarks that do not require Fortran, and report average results between them. Results per benchmark were not reported due to space limitations.

## *B. ReCycling*

Our evaluation considers the addition of extra pipeline stages to each design. Pipeline stages are added to the blocks with the worst delay. We assumed perfect recycling/retiming (perfect balancing of delays). Although this is usually not possible, this approximation is sufficient. It is only necessary to ensure that after the insertion of a pipeline stage, the two resulting stages have a delay smaller than the second most critical path before insertion. We add 2FO4 delay per added stage to account for the register overhead.

To decide which pipeline stages are unbalanced, we use synthesis results for the FPU and previously published data from FabScalar [19] that reports pipeline stage breakdowns. The minimum pipeline configuration is the original in the non-elastic baseline: 6 for FPU and 13 for the core.

Since ReCycling changes both throughput (IPC) and timing, the performance metric used is $throughput \times frequency$ (equivalent to IPS). Also, it has been shown that unless power is considered, the ideal pipeline for a design is extremely deep [20]. Therefore, we consider ED. We estimated power from synthesis results for the FPU and ESESC simulations (based on McPAT [21]) for the core, and observe that the logic energy consumption (both dynamic and leakage) remains roughly constant. However, the dynamic clock energy consumption increases linearly with both frequency and number of registers, and the leakage clock energy increases linearly with the number of registers.

## VII. Evaluation

We first show the design space exploration of the different approaches. In particular, we show that Fluid Pipelines are able to push the Pareto frontier towards better performance and energy efficiency (Section VII-A). Then, we report the more detailed results, such as the maximum frequency, throughput, and ED for different pipeline configurations for both the FPU (Section VII-B) and Out-of-Order core (Section VII-C).

## *A. Overall Results*

Fluid Pipelines push the design space towards more energy efficiency and better performance. This is accomplished by avoiding false dependencies between concurrent paths. For most of the design points in the design space, Fluid Pipelines improve both better performance and energy. In comparison, LI-BDNs reach better performance than SELF, but at the cost of more energy (and area, not evaluated here).

The Pareto frontier (Figure 13) shows that for OoO core, Fluid Pipelines (FP) deliver both less energy and more performance than SELF. Also, Fluid Pipelines improve the best performance (by 6%, but with 28% less energy) and the best energy point (by 14%, but with 16% more performance). Each point represents a different pipeline configuration, where deeper pipelines tend to improve performance while consuming more energy. In this case, LI-BDN was not used, as it will be explained in the detailed evaluation.

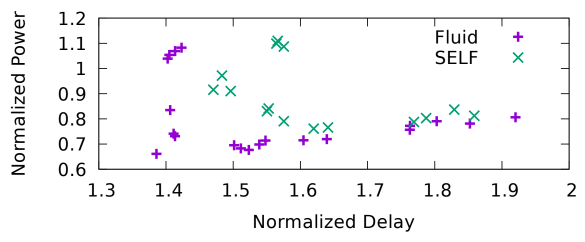


**Fig. 13:** Fluid Pipelines push the Pareto frontier for the OoO core by improving both performance and energy.

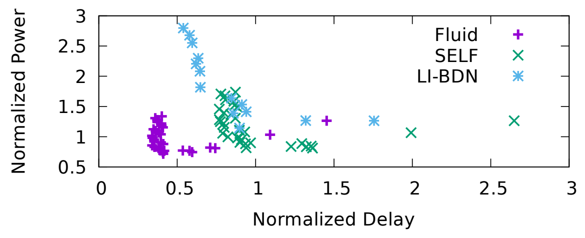


**Fig. 14:** Fluid Pipelines push the Pareto frontier for the FPU by improving both performance and energy.

For the FPU (Figure 14), LI-BDNs result in increased energy consumption due to the increased storage, but improved the performance, when compared to SELF. Fluid Pipelines present the best performance and energy out of the three schemes, since they do not require extra storage. Compared to SELF, Fluid Pipelines improve the best performance by 120%, with 21% less energy, or improve the best energy by 12% with 230% improvement in performance. In comparison with LI-BDNs, Fluid Pipelines improved the best performance by 33%, using 83% less energy, or improved the best energy by 38% with 118% better performance.

## *B. Elastic FPU*

The maximum throughput for each of the models is summarized in Table II. Fluid Pipelines deliver constant throughput regardless of the number of pipelines. The throughput of SELF decreases when there is additional pipeline stages in the sequential loops. In the case of LI-BDNs, the extra buffering helps maintaining the throughput even after the insertion of a few stages in the loops, but after a certain number of insertions, there is back pressure due to the dependencies.

The effective frequency, calculated for the average throughput, is reported Figure 15. It does not necessarily increase with the number of pipeline stages. This is due the fact that despite the frequency gain with the new pipeline stage, the reduced throughput reverts the gains and reduces the overall performance. Since in the average case the loop path is used, there is a reduction in the gap between Fluid Pipelines and the other models. The same fact also causes reduction in the throughput of both SELF and LI-BDN. Despite the reduction in the gap, Fluid Pipelines are still able to deliver a considerably improved performance compared to SELF (120%), and slightly improved performance compared to LI-BDN (40%), but using less resources.

ED is reported in Figure 16. The energy overhead caused by the extra storage in LI-BDNs reverses the advantages when compared to SELF. When comparing Fluid Pipelines with SELF, Fluid Pipelines improve the best ED point by improving performance by 176%, with 5% better energy. Alternatively, Fluid Pipelines deliver 120% better top performance (with 21% less energy). When we compare Fluid Pipelines with LI-BDNs, Fluid Pipelines improve the best ED point by improving both performance (by 163%) and energy (by 25%).

## *C. Elastic OoO Core*

LI-BDNs were not considered, since their main improvement over SELF is the addition of FIFOs between modules. This is an important overhead for both area and power. In addition, we note from the previous experiment that for deep pipelining, LI-BDN behavior approaches that of SELF.

As in the FPU case, the effective frequency fluctuates (Figure 17) when the frequency improvement is not enough to compensate for the throughput decrease. Note that for some

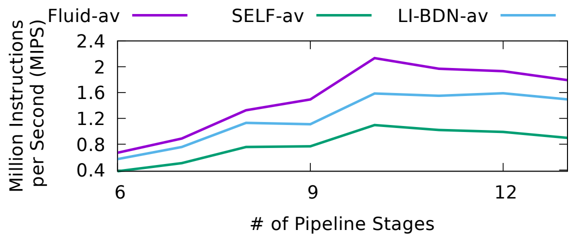


**Fig. 15:** In Fluid Pipelines, circuits can be recycled with higher throughput then possible with Elastic Systems, and thus for better system performance.

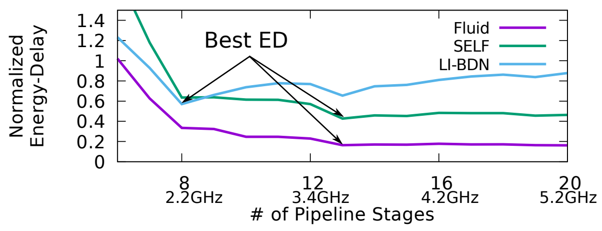


**Fig. 16:** Fluid Pipelines improve the best ED point of the FPU, pushing the depth of the pipeline.

**TABLE II:** Fluid Pipelines deliver constant maximum throughput, regardless of the number of pipeline stages.

| Pipeline stages | Fluid Pipelines | SELF | LI-BDN |
|---|---|---|---|
| 6 | 1 | 1 | 1 |
| 7 | 1 | 1 | 1 |
| 8 | 1 | 1 | 1 |
| 9 | 1 | 0.67 | 1 |
| 10 | 1 | 0.50 | 1 |
| 11 | 1 | 0.40 | 0.83 |
| 12 | 1 | 0 37 | 0.74 |
| 13 | 1 | 0.33 | 0.67 |

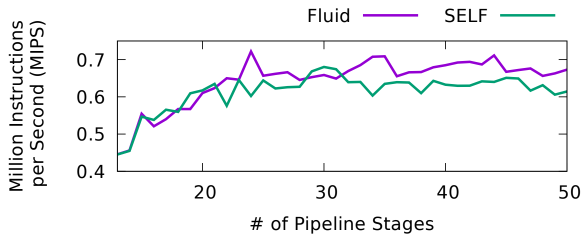


**Fig. 17:** Effective frequency alone is not a fair metric since it does not consider the extra registers added by SELF.

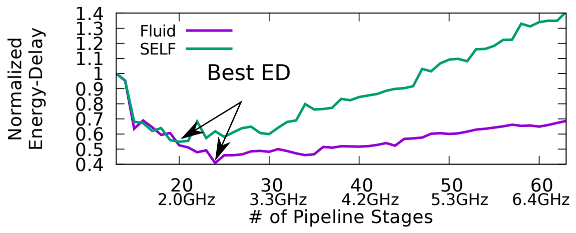


**Fig. 18:** Fluid Pipelines shift the optimal ED point of the pipeline depth for a complex OoO core, and improve performance with a smaller power overhead.

points, SELF yields better overall performance than Fluid Pipelines. This is due the insertion of extra pipeline stages into all the paths that are parallel to the critical path, which in some cases ends up hitting the second most critical path, and yields a better frequency increase, with a cost in power and area (area is not reported).

The first few stages added increase the frequency considerably, with relatively small hit on IPC (throughput) and energy. This leads to an improvement in the ED. As the pipeline depth increases, the addition of extra stages has a smaller impact on frequency, but lowers IPC more. In other terms, a relatively high number of stages (*i.e.*, power overhead) is needed to improve the overall performance, and thus ED gets worse. In SELF, when one stage is added to a path, the optimal solution for throughput is to also add a stage in all parallel paths with extra power overhead. Also in SELF, adding stages has a negative effect on throughput. Combining these two effects results in a faster degradation of ED. Fluid Pipelines shift the optimal number of pipeline stages, make a deeper pipeline configuration, while improving energy by 13% and performance by 17%.

## VIII. Conclusion

A new abstraction for Elastic Systems, Fluid Pipelines, is proposed. By using Fluid Pipelines, the designer has the opportunity to extract OoO execution from the circuit whenever possible, and boost the design performance. Fluid Pipelines push the design's Pareto frontier, by improving performance and energy. In our experiments, Fluid Pipelines improve the optimal ED configuration of an OoO core by improving energy 13% and performance by 17%, over SELF. For a pure high performance configuration, Fluid Pipelines deliver 6% better top performance while using 28% less energy.

We present a modeling framework using Coloured Petri Nets, which allows us to evaluate the system runtime behavior, and perform early design space exploration. This framework is used to evaluate Fluid Pipelines against other Elastic System approaches, showing an improvement in the overall throughput of the systems. We argue for the use of this simple tool when evaluating simple event-driven systems.

Fluid Pipelines open many research opportunities in EDA and architecture like automatic pipeline transformations. It can benefit from new DSLs for hardware description. Our future work includes RTL and gate level evaluation of the proposed model and transformations which, in turn, leads to a better understanding of the overheads of this new technique as well as the design trade-offs in terms of area and power.

## Acknowledgments

This work was supported in part by the National Science Foundation under grants CNS-1059442-003, CNS-1318943-001, CCF-1337278, and CCF-1514284. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF.

## References

[1] J. Cortadella, M. Galceran-Oms, and M. Kishinevsky, "Elastic Systems," in *Proc. of the 8th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign*, 2010.

[2] J. Cortadella, M. Kishinevsky, and B. Grundmann, "SELF: Specification and Design of Synchronous Elastic Circuits," in *Proc. of the ACM/IEEE International Workshop on Timing Issues*, 2006.

[3] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Performance analysis and optimization of latency insensitive systems," in *Proc. of the 37th Design Automation Conference*, 2000.

[4] M. Vijayaraghavan and A. Arvind, "Bounded Dataflow Networks and Latency-Insensitive Circuits," in *Proc. of the 7th IEEE/ACM Int'l Conf. on Formal Methods and Models for Codesign*, 2009.

[5] D. Bufistov, J. Cortadella, M. Galceran-Oms, J. Julvez, and M. Kishinevsky, "Retiming and recycling for elastic systems with early evaluation," in *46th Design Automation Conference*, 2009.

[6] C. E. Leiserson and J. B. Saxe, "Retiming Synchronous Circuitry," *Algorithmica*, vol. 6, pp. 5–35, 1991.

[7] J. Julvez, J. Cortadella, and M. Kishinevsky, "Performance analysis of concurrent systems with early evaluation," in *Computer-Aided Design. Int'l Conf. on*, Nov 2006.

[8] B. Cao, K. Ross, M. Kim, and S. Edwards, "Implementing Latency-Insensitive Dataflow Blocks," in *Proc. of the 13th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign*, Jul 2015.

[9] R. T. Possignolo, E. Ebrahimi, H. Skinner, and J. Renau, "FluidPipelines: Elastic circuitry without throughput penalty," in *Logic Synthesis (IWLS), Proc. of the 2016 International Workshop on*, Jun 2016.

[10] K. Jensen and L. M. Kristensen, *Coloured Petri Nets Modelling and Validation of Concurrent Systems*. Springer-Verlag Berlin Heidelberg, 2009.

[11] D. Baudisch and K. Schneider, "Evaluation of speculation in out-of-order execution of synchronous dataflow networks," *Int. J. Parallel Program.*, vol. 43, no. 1, pp. 86–129, Feb. 2015.

[12] M. Oskin, F. Chong, and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs," in *International Symposium on Computer Architecture*, Vancouver, Canada, Jun. 2000, pp. 71–82.

[13] L.-F. Chao, A. LaPaugh, and E.-M. Sha, "Rotation scheduling: a loop pipelining algorithm," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 16, no. 3, pp. 229–239, Mar 1997.

[14] G. Dimitrakopoulos, I. Seitanidis, A. Psarras, K. Tsiouris, P. M. Mattheakis, and J. Cortadella, "Hardware primitives for the synthesis of multithreaded elastic systems," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2014, pp. 1–4.

[15] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu, "Elastic CGRAs," in *Proc. of the ACM/SIGDA Int'l Symp. on Field Programmable Gate Arrays*, 2013.

[16] L. P. Carloni, K. McMillan, A. Saldanha, and A. Sangiovanni-Vincentelli, "A Methodology for Correct-by-construction Latency-insensitive Design," in *Computer-Aided Design. Int'l Conf. on*, 1999.

[17] I. Ganusov, H. Fraisse, A. Ng, R. T. Possignolo, and S. Das, "Automated extra pipeline analysis of applications mapped to Xilinx UltraScale+ FPGAs," in *Field Programmable Logic and Applications (FPL), Proc. of the 26th Conference on*, Aug 2016.

[18] "Opencores.org," http://opencores.org/.

[19] N. Choudhary, B. Dwiel, and E. Rotenberg, "A physical design study of FabScalar-generated superscalar cores," in *VLSI and System-on-Chip (VLSI-SoC), 2012 IEEE/IFIP 20th International Conference on*, Oct 2012.

[20] M. Hrishikesh, D. Burger, N. P. Jouppi, K. I. Farkas, and P. Shivakumar, "The optimal logic depth per pipeline stage is 6 to 8 fo4 inverter delays," *Proc. of the 29th. Int'l Symp. on Computer Architecture*, 2002.

[21] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture. 42nd IEEE/ACM Int'l Symp. on*, 2009.