Fluid Pipelines: Elastic Circuitry without Throughput Penalty

Rafael Trapani Possignolo
rafaeltp@soe.ucsc.edu

Elnaz Ebrahimi
elnaz@soe.ucsc.edu

Haven Skinner
hskinner@soe.ucsc.edu

Jose Renau
renau@ucsc.edu

University of California, Santa Cruz
1156 High Street
Santa Cruz, CA, USA
http://masc.cse.ucsc.edu

ABSTRACT

In chip design, pipeline depth and cycle time are fixed early in the design process but their impact on physical design can only be assessed when the implementation is mostly done, making it impractical to change such parameters. Elastic Systems are insensitive to latency, and thus enable changes in the pipeline depth late in the design time with low effort. Nevertheless, current Elastic System implementations have significant throughput penalty when stages are added in the presence of pipeline loops. We propose Fluid Pipelines, an evolution that allows pipeline transformations within an Elastic System without throughput penalty. Formally, we introduce "or-causality" in addition to the already existing "and-causality" in Elastic Systems. This gives more flexibility than previously possible but requires the designer to annotate the intended behavior of the circuit. Fluid Pipelines are able to improve the optimal energy-delay (ED) point by shifting both performance (by 176%) and energy (by 5%). Fluid Pipelines also allow for exploration of the Pareto frontier enabling points like delivering 33% better top performance using 83% less energy. Fluid Pipelines open many research opportunities in both EDA and architecture and enable interesting design space exploration. We envision a scenario where automated tools would be able to generate, from the same RTL different pipeline configurations for, *e.g.*, low power, high performance, so forth. In that sense, Fluid Pipelines are able to greatly reduce design effort.

1. INTRODUCTION

In current digital design practices, cycle time and pipeline depth are set early in the design process due to their impact on the other design parameters. Meeting a certain cycle time usually requires multiple time-consuming iterations between design and implementation [11]. Elastic (or latency insensitive) Systems [5, 7, 8, 18] are an alternative to the traditional fixed cycle pipeline paradigm. Elastic systems are based on the assumption that the correctness of the system does not depend on the latency (number of clock cycles) between two subsequent events, but on their order [5, 18]. This allows for the insertion of new stages later in the design time without breaking the circuit correctness [5].

Changing the number of pipeline cycles, also known as Recycling [2, 16], is possible in Elastic Systems but constrained by the presence of sequential loops[1]. This reduces the applicability of recycling, because most complex circuits, such as processors, include sequential loops. Traditional Elastic Systems rely on an automated flow that transform regular synchronous circuitry into elastic. Since the flow does not have knowledge on the intended behavior of the circuit, it has to maintain the completion order of events. This has the side effect of reducing the overall throughput of the circuit [5, 15]. Throughput losses can be mitigated by the use of Early Evaluation [2] but the whole system remains limited by the worst sequential loop, even if such loop is not actually used.

In contrast, Out-of-Order (OoO) execution is omnipresent in modern digital design and is known to improve system throughput. In this paper, we propose Fluid Pipelines, an evolution of current Elastic circuitry, that enable unordered completion order. Since the flow cannot change the behavior of a circuit,Fluid Pipelines rely on designer annotations to 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 [3], and is thought to improve design methodologies [18]. We go one step further and evaluate new design paradigms within Elastic Systems, one that allow for OoO behavior. By exposing the Merge and Branch to the designer, the flow becomes aware of the intended behavior, and when the relative completion between operations does not affect design correctness.

Fluid Pipelines are able to re-claim the throughput losses from the automated conversion. The automated flow of Elas-

[1]By sequential loops, we mean cycles in the graph representing the connections between registers, it should not be confused with program loops.

tic 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 the inputs of multiple different stages, whereas Join is used when parallel data paths re-unite, and thus the inputs of a single stage come from multiple separate stages. The Join operator requires all the inputs to be valid to proceed. Typical examples are the inputs of an adder unit that need to be present 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 since it is triggered when at least one of the inputs has valid data (and thus has "or-causality"), also, only data from one of the inputs is consumed at each cycle. Its dual, Branch, is an operator that propagates data to only one of multiple output paths, as opposed to sending data to all the output paths as Fork.

This behavior is found in many places in digital system designs. For example, a Floating Point Unit (FPU) has parallel paths for additions, multiplications, divisions, etc. Each path is triggered independently of the others. Another example is a network router, where independent packages come from different inputs, and propagates to a single output (based on the destination and routing policy).

To evaluate the performance of Fluid Pipelines, we propose a new methodology that can evaluate Fluid Pipelines and traditional Elastic Systems. We model Fluid Pipelines and Elastic systems using Coloured Petri Nets (CPN) [14] to determine the overall performance, which considers both throughput and frequency. This is then used to find the optimal pipeline configuration for a given design. This performs faster than RTL simulation for all possible pipeline configurations.

We observed that the presence of loops in the test cases for the traditional elastic approach dramatically deprecated the throughput, and consequently the system performance. Contrarily in Fluid Pipelines, the increase of circuit frequency resulted in a performance increase up to twice the frequency. The Fluid Pipelines area overhead is virtually zero compared to the traditional elastic approach. Our results show improvements of up to 176% in performance, and 5% lower power. With the use of CPN models, it is possible to explore the Pareto frontier and select other interesting design points, depending on the specific application, but also to have a more fair comparison between Elastic Systems.

The contributions of this paper are:

- Fluid Pipelines (Section 4), an Elastic System evolution that improves the Pareto frontier by avoiding typical throughput loss typical in traditional Elastic Systems.
- A new evaluation methodology (Section 5) using Coloured Petri Nets for Elastic Systems and Fluid Pipelines.
- An evaluation (Section 7) of a FPU to quantify the impact of traditional Elastic Systems and Fluid Pipelines.

2. RELATED WORK

Out-of-Order and Speculation have been evaluated in software Dataflow Networks [1]. Dataflow network concepts are used for task scheduling in parallel execution. The proposal relies on speculating what dependencies are real or false dependencies, thus need to trigger re-execution when a misspeculation happens. In our approach, we rely on the designer knowledge of the logic to avoid such scenario.

High Level Synthesis (HLS) [17] is a technique to synthesize the code written in programming languages, as opposed to Hardware Description Languages (HDLs). Using HLS, designers can put effort into functional design, while tools take care of pipelining in a process called scheduling. For instance, Chao et al. [6] propose a scheduling algorithm capable of retiming and reducing the pipeline depth in loops, thus it partially performs recycling. To some extent, HLS attacks the same problems as Fluid Pipelines, but Fluid Pipelines propose a lower level approach, giving the designer a more fine-grained control over the final design. HLS could leverage Fluid Pipelines under the hood to enable recycling in such loops, and in that regard, Fluid Pipelines and HLS can be viewed as orthogonal techniques. In fact, that could improve design time in HLS, because it could enable changes in the pipeline configuration without the need to regenerate RTL.

A formal approach to asynchronous logic has been proposed [10]. This approach expresses logic operators as a function of control signals (equivalent to those used in Elastic circuits). The paper provides good insights on logic optimizations considering 4 logic values (0, 1, NULL, busy), but there is considerable overhead since each logic gate depends on control signals. The formal approach proposed could be adapted to a coarser grain, and thus, could be used in the context of Elastic Systems.

In the context of Elastic Systems, different approaches have been proposed to mitigate the throughput loss due to the presence of sequential cycles. The use of an *Eager Fork* operator [8] lets one of the paths to start executing even if the parallel path still has its stop signal active. Nevertheless, semantics are not changed, and thus Eager Fork has to wait until the second path resets the stop bit. This becomes problematic when only one of the paths is used, and the other path takes a few cycles to reset the stop signal. In such cases, the backpressure will propagate to the stages that precede the Fork. Fluid Pipelines are designed to avoid such scenarios by not waiting for parallel paths when there are no dependencies between them.

Early Evaluation [2] has been proposed in the SELF framework [8]. It is a more sophisticated mechanism that determines which inputs in merging paths are actually needed (for instance, in a mux), and only waits for the inputs that are actually needed. The next token in the remaining inputs will be ignored to maintain correctness. This can be implemented as a counter that keeps the track of how many tokens need to be dropped, or as a back propagation of anti-tokens that annihilate the first token they encounter. Early Evaluation has been shown to improve the throughput in Elastic Systems in the average case. The main issue is that even if a sequential loop is not currently being used, it still limits the overall system throughput.

LI-BDNs [18] are a generalized form of Elastic Systems. They use First In First Out (FIFO) queues as the communication channel between modules. The main advantage is that, the use of FIFOs creates natural clock region boundaries. In LI-BDN systems, a stage-global enable signal is used for all

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

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

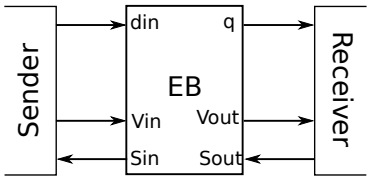


Figure 2: Elastic buffers are the basic construct blocks of Elastic Systems and can be viewed as queues with a limited size.

the state elements (registers). Therefore, a synchronous circuit can be transformed into *Patient Circuits* by "and"-ing enable signals in all the registers of each module with the global module enable. The global module enable signal indicates a *stall* event. The increased buffering capacity due to the presence of FIFOs improves the overall performance, at the cost of more area overhead. Fluid Pipelines performance is better than LI-BDNs and uses less area.

3. BACKGROUND

Elastic System is a system whose functionality only depends on the order of its inputs and not their exact arrival time [4]. An elastic execution example is shown in Figure 1, 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 cannot be used anymore. Empty cells in the table denote that no new data arrived in that cycle. Note that the latency between events is arbitrary.

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* ($V = 1, S = 1$) [8].

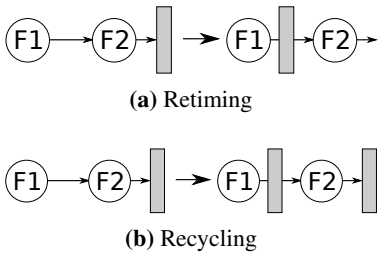
Elastic Buffers (EBs) are storage units that replace registers, they include handshake signals both in the input and output interface. Figure 2 shows the interface of an EB with input and output control signals.

3.1 ReCycling and Retiming in Elastic Systems

To improve the frequency or decrease the area of Elastic Systems, it is possible to move EBs across circuit blocks (Retiming) [2] (Figure 3a), or to insert additional stages in long wires [5] or in between combinational logic (ReCycling) [2] (Figure 3b). Retiming preserves the sequential behavior of the circuit [2] and thus it can be applied to any type of circuit mostly without penalties.

In the case of recycling, Júlvez *et al.* [15] observe that the throughput of a system is limited to the sequential loop with

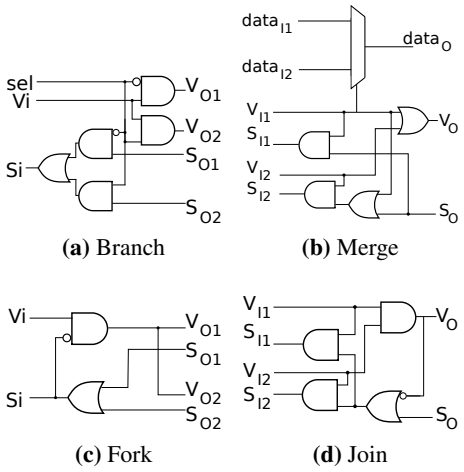
[2]Other equivalent naming conventions have been used, for instance, Elasticity has been expressed in terms of FIFO operation [18].



(a) Retiming

(b) Recycling

Figure 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

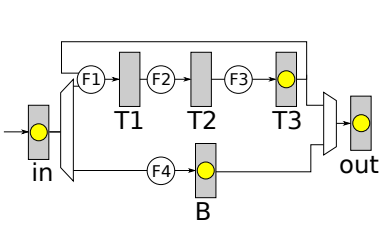
Figure 4: Fluid Pipelines uses different operators to indicate the intended functionality of a circuit and enable better design space exploration. Branch and Merge are used when the relative order of operations can be broken, while Forks and Joins enforce ordering. Note the difference in the handling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

lowest throughput, and that the throughput of a sequential loop can be calculated as the number of tokens in the loop divided by the number of EBs in the loop. The throughput of a cycle can increase with Early Evaluation depending on how often each event occurs [15], but due to back-pressure, there is still a limit on such mitigation. ReCycling is able to reduce cycle time, but may reduce throughput in the case of stage insertion in sequential loops [2, 5, 15].

4. Fluid Pipelines

Fluid Pipelines evolves the traditional Elastic Systems to allow breaking the relative completion order. To implement this behavior, Fluid Pipelines uses four types of operators: *Branch*, *Merge*, *Fork* and *Join* operators (Figure 4) [9]. In Figure 4a, *Selection* (sel) is a data-dependent selection signal that indicates to which output the data will propagate to. The operators can be easily extended to more than 2 inputs/outputs.

Branch is used when the datapath forks into multiple paths, but data should propagate to only one of them, this choice is data dependent and 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. The Merge operates as an arbiter: multiple senders compete for a single output.



(a) Loop Case

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

(b) Instructions

Figure 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.

The sender that wins the arbitration propagates data. In our FPU example, a Merge would be used at the output of the functional units when results from each unit are collected. Another way to think of the Merge is to notice 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. Disjoint or-causality permits low latency arbitration [19]. For simplicity and without loss of generality, the proposed implementation in Figure 4b has fixed priority and could cause starvation, but it 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.

As conceptual example of the power of Fluid Pipelines, 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 1.

The execution order of Fluid Pipelines is altered (Table 1), note how in cycle 3, it is possible to move I3 to the bottom path, while the top path is still executing. This re-ordering is a result of the "or-firing" rule. This is fine, because that behavior was specified by the user, and not arbitrarily changed by the tool. In a processor core, the reordering buffer is already doing such function, while in network-on-chips, the re-ordering is usually not performed. Since this requirement is application specific, it is left out of this manuscript. We assume that, if needed, the re-ordering is performed in the design. In the case where order should be maintained, regular Fork and Join operators (defining "and-firing" rules) must be used, which cause the design to behave similar to a regular Elastic System.

Table 1: Sample trace for the toy case, Fluid Pipelines deliver higher throughput then Traditional Elastic.

	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						

4.1 Fluid Pipelines Deadlock Avoidance

In loop structures, deadlocks are a concern. Vijayaraghavan and Arvind [18] show that, in Elastic Systems, deadlocks come from extraneous dependencies, *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 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 as a Fluid Pipelines circuit. There are multiple options for that.

The most straightforward implementation of the block follows the behavior described in Figure 7, which waits until all the inputs have valid data, and until all the outputs can accept new data to perform the operation. This 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. This is a violation to the no extraneous dependencies directive.

A simple solution to this case is the use of a Fork operator. The Fork operator isolates the handshake handling, and thus

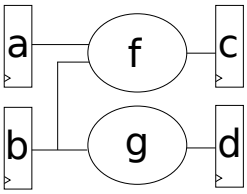


Figure 6: Fluid Pipelines design uses a few design practices to avoid deadlocks. Those are restriction on how to implement a given design and not on which designs can be implemented.

```
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;
```

Figure 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
endmodule

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;
endmodule
```

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

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. Consider a circuit that produces n inputs per token consumed. Now, let the output of this circuit be connected back to its input. For a buffer with size m it is clear that after m/n cycles, the buffer will be full, and a deadlock situation will arise.

5. NEW EVALUATION METHODOLOGY

In order to find the optimal pipeline depth, a designer or an automated tool must estimate the throughput of a given pipeline configuration (*i.e.*, number and position of pipeline stages). To estimate the throughput of Fluid Pipelines designs, we propose the use of Coloured Petri Nets (CPN) [14]. CPNs are used for design specification and evaluation based on events/transitions and data/tokens which is what we need to perform Branch-like operations.

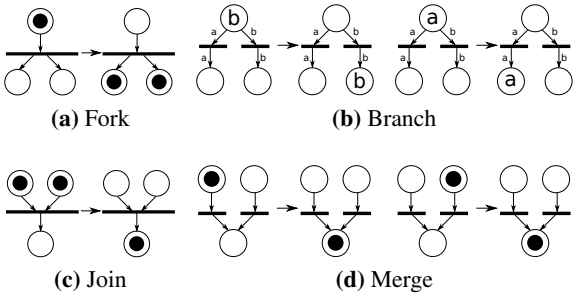
Petri Nets can be defined as a bipartite graph of *places* and *transitions*, connected by *arcs*. Places can contain *tokens* and tokens have data value attached to them. The attached value is the token *colour*. The state of the net (the *marking*) is defined by the number of tokens in each place. The initial marking is changed when transitions *fire*. When a transition fires, tokens are subtracted from its input places and are added to its output places according to *arc expressions*. There is a *capacity* associated with each place that represents the maximum number of tokens in that place, and prevents input transitions from firing (note that this is not part of the original formulation of PNs, but has been proposed as an extension). In CPNs, the tokens are typed (*colour*), and transitions are type-dependent.

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.*
- Σ *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 to each place a maximum capacity.*

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 $G(t)(M) = true$ (*i.e.*, the guard functions are satisfied), and $\forall a \in \{b|b = (p,t), p \in P, b \in A\}, E(a)(M) <= M(a.p)$ (*i.e.*, each input place contains the appropriated tokens), and $\forall p \in t\bullet, M(p) < Cap(p)$, where $t\bullet = \{p|(t,p) \in A\}$ (no output place is "full"). 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,



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

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

Figure 9: CPN models can be used to estimate the overall throughput of Fluid Pipelines and Elastic Systems.

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.

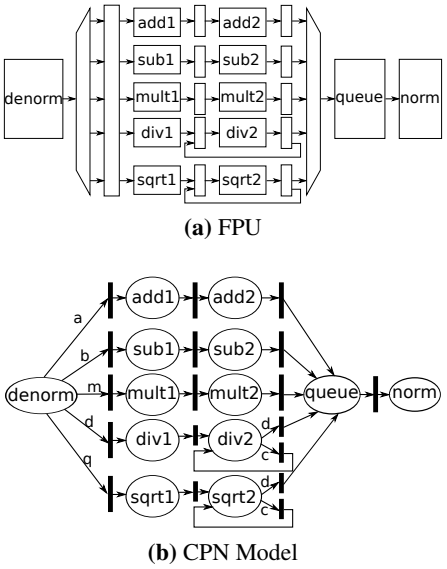
Figure 9 depicts how the Fluid Pipelines' operators are modeled as CPN transitions. Circles represent places, bars represent transitions, and dots represent tokens in transitions that are not colour dependent while letters represent coloured tokens. In case of Merge operators, the semantic does not define which transition has priority, and thus, conceptually they 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, a mapping can be defined between the guard functions and the handshaking logic.

6. EVALUATION SETUP

To evaluate Fluid Pipelines, we consider a fully compliant IEEE-754 in-house FP Unit, designed both as synchronous (for previous approaches), and annotated with Fluid Pipelines operators. It has a simple structure, but is sufficient to demonstrate how Fluid Pipelines can yield better performance compared with previous Elastic Systems.

A functional block diagram of the FPU unit is presented in Figure 10a. This is a simple structure with multiple parallel paths and simple loops. The CPN model used for the performance evaluation is shown in Figure 10b, considering Fluid Pipelines. In this case, the Merge and Branch operators are used. Note how the division and square root modules use the Branch 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.

Fluid Pipelines are compared against SELF [2] and LI-BDNs [18]. 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) FPU

(b) CPN Model

Figure 10: CPN modeling can be used to evaluate system performance.

6.1 ReCycling

Our evaluation considers the addition of extra pipeline stages to each design. Pipeline stages are always 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 good enough[3]. 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.

The performance metric used is $throughput \times frequency$ (equivalent to IPS), since ReCycling changes both IPC and timing, thus those are combined. Also, it has been shown that unless power is considered, the ideal pipeline for a design is extremely deep [12, 13]. Thus, we consider energy-delay (ED). We 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.

[3]The requirement is that the delay on each one of stages after the split will be smaller than the delay of the second most critical path. For instance, say the critical path in stage 1 has a delay of 1ns, whereas the critical path in stage 2 has a delay of 0.7ns. We want to add a new register to stage 1 such that the delays of the newly generated stages are less than 0.7ns, but perfect balance between the stages is unnecessary.

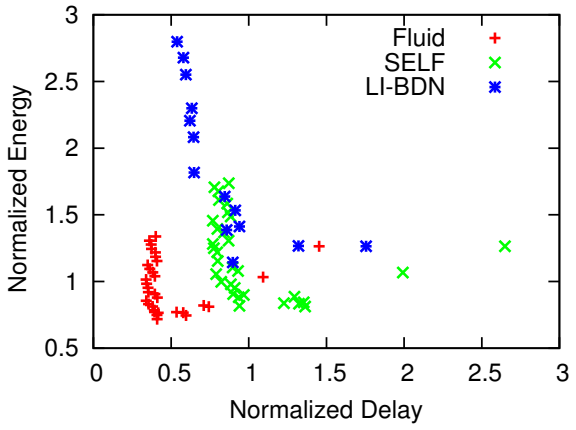


Figure 11: Fluid Pipelines are able to push the Pareto frontier for the FPU by improving both performance and energy.

7. EVALUATION

We start our evaluation by showing 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. Then, we proceed to explain the detailed results, such as the maximum frequency, throughput and energy-delay for different pipeline configurations for the FPU.

Fluid Pipelines push the design space towards more energy efficiency and better performance. This is mostly accomplished by avoiding false dependencies between concurrent paths. For most of the design points, Fluid Pipelines were able to deliver both better performance and energy. When comparing SELF with LI-BDNs, the former was able to obtain better performance, but at the cost of energy (and area, which was not evaluated here).

The Pareto frontier (E vs D) is shown in Figure 11). LI-BDNs result in increased energy consumption due to the increased storage, but are able to improve the performance, when compared to SELF. Fluid Pipelines present the best performance and energy out of the three schemes, since it does not requires extra storage. When compared to SELF, Fluid Pipelines were able to improve the best performance by 120%, with 21% less energy, or improve the best energy by 12% with 230% improvement in performance. When compared to LI-BDNs, Fluid Pipelines improved the best performance by 33%, using 83% less energy, or improve the best energy by 38% with 118% better performance.

The results show that Fluid Pipelines designs are both more energy efficient and higher performance than designs possible with current Elastic Systems.

7.1 Detailed Results

The maximum throughput for each of the models is summarized in Table 2. This is calculated by the use of a synthetic workload that only considers the best path (addition, subtraction and multiplier in this case). The initial pipeline depth in the design is 6, thus, there is no data for any configuration with less stages than 6. Fluid Pipelines are able to 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.

Table 2: Maximum FPU throughput for the different evaluated models. 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

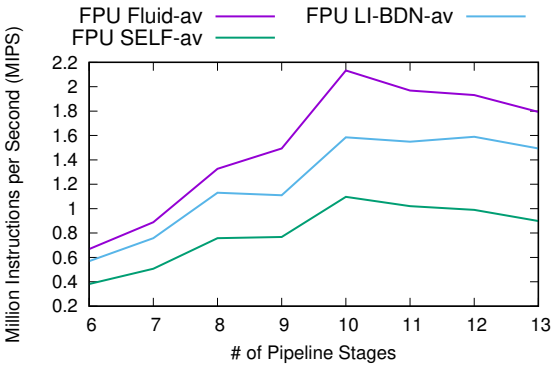


Figure 12: In Fluid Pipelines, circuits can be recycled with higher throughput then possible with Elastic Systems, and thus for better system performance.

Maximum throughput is not a realistic metric. Thus, we calculate the average throughput over a million random instructions, we then report the effective frequency, shown in Figure 12. Note that effective frequency 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.

To take into account the extra stages added in the case of SELF, we use energy-delay product (ED), that considers both performance and energy. These numbers are reported in Figure 13. 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 are able to improve the best ED point by improving performance by 176%, with 5% better energy. Alternatively, Fluid Pipelines are able to deliver 120% better top performance (with 21% less energy). When we compare Fluid Pipelines with LI-BDNs, Fluid Pipelines improve the best

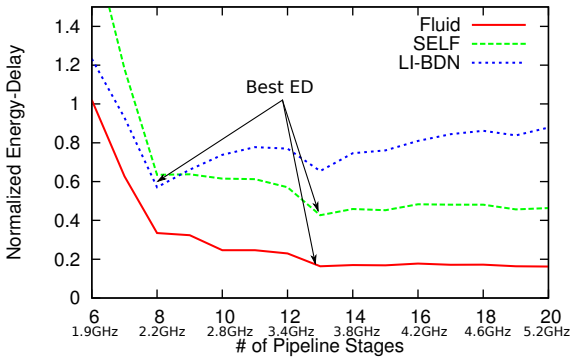


Figure 13: Fluid Pipelines are able to improve the best ED point of the FPU, pushing the depth of the pipeline.

ED point by improving both performance (by 163%) and energy (by 25%). Fluid Pipelines are also able to deliver 33% better top performance (with 83% less energy).

8. CONCLUSION

A new abstraction for Elastic System, Fluid Pipelines, was proposed. By using Fluid Pipelines, the designer has the opportunity to extract out-of-order execution from the circuit, whenever possible, and thus boost the design performance. Fluid Pipelines push the Pareto frontier of designs, by improving both performance and energy. In our experiments, over SELF, Fluid Pipelines improve the optimal energy-delay configuration of a FPU design by improving energy by 5% and performance by 176%. Alternatively, Fluid Pipelines were able to reduce the lowest energy point by 12%, with 120% better performance, or improve the top performance by 33% (but with 83% less energy).

We present a modeling framework for the proposed abstraction, using Petri Nets, which allows us to evaluate the system run-time behavior, and is a powerful tool for early design space exploration of Fluid Pipelines. 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 repipelining with larger codes than the evaluated FPU. Fluid Pipelines can also benefit from new DSLs for hardware description, and further work to include RTL and gate level evaluation of the proposed model and transformations which in turn leads to a better understanding of the overheads of the new technique, as well as better understanding of the design trade-offs in terms of area and power.

Acknowledgments

We like to thank the reviewers for their feedback on the paper. 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.

9. REFERENCES

[1] D. Baudisch and K. Schneider. Evaluation of speculation in out-of-order execution of synchronous dataflow networks. *Int. J. Parallel Program.*, 43(1):86–129, Feb. 2015.

[2] 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*, pages 288–291, 2009.

[3] B. Cao, K. Ross, M. Kim, and S. Edwards. Implementing Latency-Insensitive Dataflow Blocks. In *Proceedings of the 13th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign*, MEMOCODE '15, Jul. 2015.

[4] 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*, pages 309 –315, 1999.

[5] L. P. Carloni and A. L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In *Proceedings of the 37th Design Automation Conference*, pages 361–367, New York, NY, USA, 2000. ACM.

[6] 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*, 16(3):229–239, Mar 1997.

[7] J. Cortadella, M. Galceran-Oms, and M. Kishinevsky. Elastic Systems. In *Proceedings of the 8th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign*, MEMOCODE '10, pages 149–158, Jul. 2010.

[8] J. Cortadella, M. Kishinevsky, and B. Grundmann. SELF: Specification and Design of Synchronous Elastic Circuits. In *Proceedings of the ACM/IEEE International Workshop on Timing Issues*, TAU 06, 2006.

[9] 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)*, pages 1–4, March 2014.

[10] K. Fant and S. Brandt. Null convention logictm: a complete and consistent logic for asynchronous digital circuit synthesis. In *Application Specific Systems, Architectures and Processors, 1996. ASAP 96. Proceedings of International Conference on*, pages 261–273, Aug 1996.

[11] S. . FGPA and S. D. S. Innovations. Altera inc.

[12] A. Hartstein and T. R. Puzak. The optimum pipeline depth for a microprocessor. In *Proc. of the 29th Int'l Symp. on Computer Architecture*, pages 7–13, Washington, DC, 2002. IEEE Computer Society.

[13] 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. *Proceedings of the 29th. Int'l Symp. on Computer Architecture*, 2002.

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

[15] J. Julvez, J. Cortadella, and M. Kishinevsky. Performance analysis of concurrent systems with early evaluation. In *Computer-Aided Design. Int'l Conf. on*, pages 448–455, Nov 2006.

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

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

[18] M. Vijayaraghavan and A. Arvind. Bounded Dataflow Networks and Latency-Insensitive Circuits. In *Proceedings of the 7th IEEE/ACM Int'l Conf. on Formal Methods and Models for Codesign*, pages 171–180, Piscataway, NJ, USA, 2009. IEEE Press.

[19] A. Yakovlev, M. Kishinevsky, A. Kondratyev, L. Lavagno, and M. Pietkiewicz-Koutny. On the models for asynchronous circuit behaviour with or causality. *Formal Methods in System Design*, 9(3):189–233, 1996.