RTL synthesis: from logic synthesis to automatic pipelining

Jordi Cortadella, Marc Galceran-Oms, Mike Kishinevsky, and Sachin S. Sapatnekar

***Abstract*—Design automation has been one of the main propellers of the semiconductor industry with logic synthesis being one of the core technologies in this field. The paper reviews the evolution of logic synthesis until the advent of techniques for automatic pipelining based on elastic timing, either synchronous or asynchronous. The emergence of these techniques can enable a productive interaction with tools that can do microarchitectural exploration of complex designs.**

I. Introduction

Since the early days of manual layout, EDA has witnessed a proliferation of abstract models, algorithms, description languages and design frameworks that have allowed to scale productivity. After the introduction of the first VLSI layout tools, designers started working with objects at a higher level of abstraction, logic gates, and logic synthesis emerged as one of the main engines to satisfy the competitive requirements of quality and short time-to-market.

Most of the innovation in logic synthesis has taken place in academia, where UC Berkeley has been one of the leading actors. Since the early 80's, logic synthesis has evolved to accommodate the increasing complexity of circuits. Today, we have specification models at the level of algorithms and transactions that enable designers to specify behaviors at abstractions levels similar to those used in software.

In this long trip, timing has always been one of the essential design parameters under optimization and a variety of timing models have been proposed to devise algorithms that can deal with performance and interfacing constraints. We can model the execution time (E) of an algorithm running for a specific data set by the following *Performance Equation*:

$$E = N \cdot P \tag{1}$$

where N is the number of cycles required to execute the task and P is the cycle period.

In combinational logic synthesis [3], [4], timing optimizations are directed to shorten the critical paths in the combinational regions and reduce P without modifying the sequential elements (latches and flip-flops) of the circuit. Sequential optimizations, such as retiming [34], permit the modification of the internal sequential behavior of the circuit and give more opportunities for timing optimizations. Still, all the optimizations preserve the external cycle-level behavior (N is not modified), thus avoiding any re-synthesis of the interfaces.

In the last decade, High-Level Synthesis [17] has evolved as a competitive alternative to manual RTL. Among different techniques for scheduling operations under performance and resource constraints, several forms of loop pipelining have been proposed to reduce N through the parallelization of operations from different iterations [22].

An essential step for timing optimization takes place when the strict timing constraints at the external interfaces are relaxed and handshaking mechanisms are introduced to determine the validity of data at the communication channels. This new paradigm is known as *timing elasticity* [11], which opens a new avenue of optimizations that can either be applied to asynchronous or synchronous circuits. With this new scenario, the optimization parameter is *average* rather than worst-case performance and automation can play a relevant role in architectural pipelining [20]. Synthesis tools for architectural exploration can be created, thus conquering a field that was only in the domain of manual design.

This paper gives a historical view and discusses the evolution of synthesis methods from combinational logic synthesis to automatic pipelining. The classical combinational and sequential logic synthesis techniques are first presented, with strict timing models that do not allow any flexibility at the interfaces. High-level synthesis is next discussed and techniques for automatic pipelining are reviewed. General models for time elasticity and handshaking schemes are finally introduced. Asynchronous and synchronous elastic circuits are two related paradigms that exploit elastic timing using different variations of handshakes. The paper continues with optimizations that can be introduced when timing elasticity is allowed and are the core for automatic pipelining at RTL. The paper finally debates about how automatic pipelining can be exploited for architectural exploration. An example shows how a non-pipelined version of the DLX CPU can be sped-up by 3x with a 20% area overhead by using automatic pipelining techniques.

II. Logic Synthesis

A. Combinational Logic Synthesis

The goal of combinational logic synthesis is to optimize the timing, area, and power of an implementation under the best models at this stage of design. This step is divided into *technology-independent synthesis*, which optimizes logic expressions to obtain low implementation costs by reducing literal counts or logic depths. and *technology-dependent synthesis*, which maps the logic expressions to a cell library of pre-characterized gates, using more accurate area, delay, and power numbers. Since the loads on logic gates are related to placement, technology-dependent synthesis is often interleaved with placement. Most combinational synthesis algorithms implicitly assume that the circuits are acyclic. Although combi-

national logic may be implemented with cyclic circuits [44], such structures are not used in mainstream designs.

Early approaches to technology-independent synthesis used exact methods for two-level minimization based on sum-of-products or product-of-sums representations [18]. Since exact logic minimization is an Σ_2^p-complete problem [51], [6], these methods were largely supplanted by heuristic methods [4]. These approaches were then generalized to multi-level logic minimization [3], with enhancements based on algebraic division and algorithmic transformations of logic expressions. Multi-level methods are a vital ingredient in optimizing typical circuits that contain 10–20 levels of logic.

The problem of technology-dependent logic optimization for area/power minimization has no known exact solution in polynomial time, and is typically solved using heuristics. Beginning with a logic decomposition using simple primitives such as 2-input NANDs and inverters, the combinational circuit is modeled as a directed acyclic graph. If this is decomposed into a forest of trees where each gate in a tree has at most one fanout, then each such tree can be mapped optimally in linear time in the number of tree vertices [30]. Heuristics have also been developed for providing practical solutions with area/power/delay trade-offs. Such technology mapping approaches suffer from structural bias, wherein the solution quality depends on the initial decomposition. Methods for overcoming this problem are summarized in [13].

From a timing point of view, the main goal of combinational optimization is to ensure that the longest path through any combinational stage meets the setup time constraint: for edge-triggered circuits, the sum of the longest combinational path delay and the setup time cannot exceed the clock period.

B. Sequential Logic Synthesis

While combinational timing optimization operates on one combinational block at a time, sequential optimizations address the memory elements within the system. These memory elements may be level-clocked transparent latches or edge-triggered flip-flops, and we will refer to them as "latches" in our discussion. Various sequential logic synthesis operations, such as state minimization and state encoding, can optimize sequential elements in a system. Such operations are primarily directed towards reducing the amount of sequential and combinational logic, and can impact circuit timing by reducing the complexity and delay of a logic implementation.

Time-borrowing between sequential stages is possible through latch-based optimizations such as retiming [34] and clock skew scheduling [19]. Retiming moves memory elements across logic while leaving unchanged the latency on all cycles and input-output paths. By moving combinational boundaries, retiming allows the clock period of a circuit to be improved without altering the externally-observable cycle-level timing. Moreover, by moving latches across logic, the number of latches in a system may be reduced.

Fig. 1(a) shows an edge-triggered sequential circuit, where the rectangles correspond to latches and the combinational elements are represented by circles annotated with the element delays. Assuming zero setup times, the clock period, corresponding to the longest combinational path between latches, is 28 units, along the path containing the shaded circles. After retiming some flip-flops along the arrows shown in the figure, the resulting circuit is shown in Fig. 1(b), where the period is 17 units. The reader can verify that this is the minimum clock period achievable by the system under retiming. Fig. 1(c) shows a solution with the minimum number of latches at the expense of increasing the cycle period to 18 units.

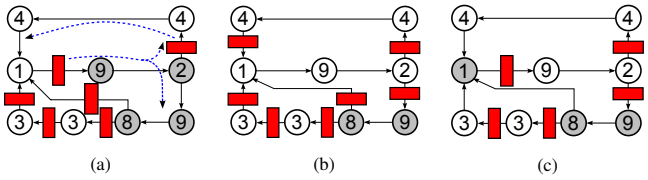


Fig. 1. A sequential circuit in (a) original form. (b) The same circuit after retiming with a (b) larger and (c) smaller number of latches.

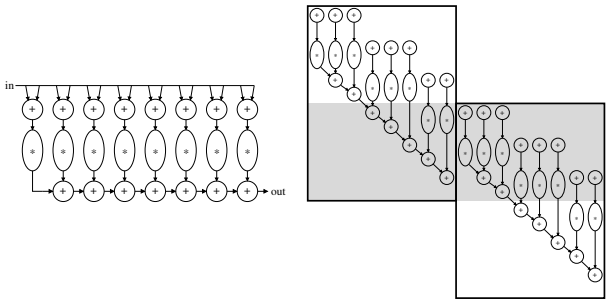


Fig. 2. Loop pipelining of a FIR filter ([24]).

The retiming problem can be formulated as an integer linear programming problem [34] that can be solved using graph and network flow algorithms. Further efficiencies are introduced in [46], [35]. Another optimization related to retiming is c-slowing [34], where each memory element is replaced by c consecutive elements. In restricted situations where unrelated data sets are processed, e.g., in acyclic pipelines, this operation can increase the circuit throughput.

III. High-Level Synthesis and Automated Pipelining

The techniques used in High-Level Synthesis (HLS) [17] aim at transforming algorithmic descriptions into RTL. Performance and resource constraints are common parameters defined by designers to provide guidelines in the generation of solutions. Optimizing performance under area constraints or optimizing area under performance constraints are typical scenarios in the exploration of the design space.

Many of the techniques used in HLS have been inherited from the body of knowledge of parallelizing compilers [23], including techniques for loop pipelining [24], [31]. The goal is to reduce E in equation (1) by reducing both N and P. The major impact of loop pipelining is in the reduction of N.

Fig. 2 depicts a classical example of loop pipelining in which the algorithmic description is shown on the left and a pipelined schedule is shown on the right. The schedule parallelizes operations from different iterations of the same loop (shaded region).

One of the main components of an optimizing compiler is the module for dependency analysis that determines the

sequencing constraints of operations. As an example, Read-After-Write (RAW) dependencies prevent operations that write and read the same operand to be re-ordered. For example, the sequence of instructions

```
R[i] = a * b;
c = R[j] + d;
```

cannot be re-ordered unless $i \neq j$. The optimizations during HLS are based on static analysis techniques. Data dependencies ara analyzed taking into account the worst-case scenarios and produce *static schedules* that are valid for any possible data. For this reason, static schedules cannot fully exploit parallelism when dependency analysis is not able to provide conclusive information.

Automated pipelining without using classical HLS techniques has been previously proposed in [32], [41]. Both methods assume that the hardware of the datapath is already partitioned into pipelined stages. Then [32] generates a stall controller and the forwarding logic, while [41] informs the designer of the available opportunities for applying forwarding and speculation and based on the designer choice generates a stalling engine. In both cases the stalling engine is a global non-pipelined controller.

In modern technology nodes, both control and datapath need to be pipelined to avoid long logic and wire delays. The elastic pipelining techniques that will be later presented in this paper allow to naturally distribute the pipeline control along the datapath. This is achieved by performing RTL transformations and using distributed handshake protocols. Elastic pipelining allows to keep track of the validity of data and define distributed pipeline boundaries that can easily interact with other components of the system. Additionally, elastic pipelining can deliver *dynamic schedules* by analyzing data dependencies "on-the-fly" and stalling the pipeline when data hazards are produced.

IV. Elastic Timing Models

The behavior of a system can be modeled as an ordered set of computations and communications that deliver some output data after reading some input data. The time elapsed between reading inputs and delivering outputs may depend on different factors: complexity of the computation, optimization of the circuit, timing of the environment, etc.

The classical methods for combinational and sequential logic synthesis preserved the cycle-accuracy of the computations, i.e., two systems were said to be equivalent if their behavior could not be distinguished externally at any cycle of the computation. This equivalence allowed to modify the internal sequential behavior of the system (e.g., by retiming flip-flops or re-encoding FSMs) but did not allow to modify the external behavior. It also enforced the environment to honor the cycle accuracy when producing the inputs and reading the outputs of the system. We refer to this model as *rigid*.

In a rigid timing model, it is not possible to break long combinational paths by introducing new pipeline stages, since this would modify the timing relationship between the inputs and the outputs of the system. However, modern systems include multiple complex components performing different computations. Very often these components work at different speeds even with clocks running at different frequencies. For the sake of modularity, it is convenient that these modules can be substituted by other instances with the same interface but different power/performance characteristics.

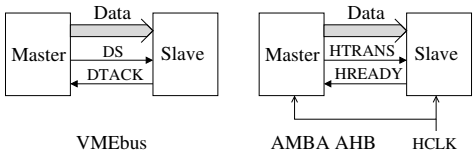


Fig. 3. Handshaking schemes for buses

To preserve this type of modularity, circuit designers have proposed systems that tolerate variable latency (aka, timing elasticity). As an example, most of the bus protocols have handshaking mechanisms to indicate when access is granted and when data is valid in the bus. Figure 3 shows two signaling schemes for standard buses, VME [25] and AMBA AHB [2]. The main difference between both is the synchronization protocol: VME is asynchronous whereas AMBA is synchronous. As in most protocols, both have two handshake signals to implement time elasticity. One signal goes from master to slave to indicate the availability of information in the bus. The other signal goes from slave to master to indicate the completion of the transfer.

In asynchronous protocols, data transfers are indicated by the events of the handshake signals and data signals are maintained stable as long as the handshake cycle has not been completed. In the VME bus, the DS signal (Data Strobe) indicates the period of time in which the master maintains valid data in the bus, while the DTACK signal (Data Transfer Acknowledge) indicates when data has been accepted by the slave.

In synchronous protocols, the initiation and completion of transfers are indicated by the value of the handshake signals at the clock edges. In the AMBA AHB bus, the HTRANS signal (two bits) may indicate an IDLE cycle when no transfer must be performed. The slave can also indicate that it has not been able to accept the transfer (HREADY=0), thus forcing the master to maintain the same data on the subsequent cycles until the transfer has been completed (HREADY=1).

Even though the elastic interfaces offer the possibility of modifying the timing at the master or slave modules, the classical logic synthesis techniques do not take advantage of this flexibility for optimization and always preserve the original timing specification at the boundaries of the modules. To exploit elasticity, the timing specifications of the modules must be manually changed at RTL and a new synthesis process must be executed. This constraint prevents automated design explorations playing with time elasticity.

A. Introducing elasticity during synthesis

Design automation requires formal models as a reference to preserve correctness when the implementation of a system is transformed by a set of synthesis rules. The incorporation of time elasticity during synthesis requires timing models that

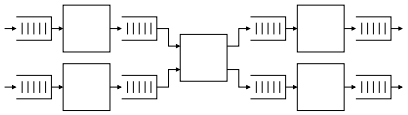


Fig. 4. Kahn Process Network.

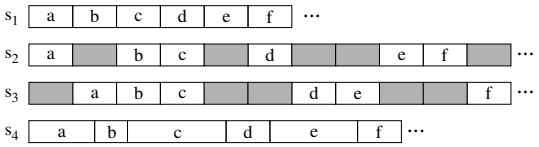


Fig. 5. Equivalent streams

go beyond the cycle-by-cycle equivalence imposed by rigid timing.

One of the simplest and most general models for elasticity is Kahn Process Networks (KPNs) [28], where a system is a set of processes that communicate through unbounded first-in-first-out (FIFO) channels, as shown in Fig. 4. Every channel can be considered as an infinite queue of tokens and the execution model is similar to the firing semantics of Petri nets [40]. A process can start a computation when data is available at all input channels. Every computation reads data from the input channels and writes data to the output channels.

With this execution model, equivalence is defined in terms of data streams in the communication channels, i.e., two systems are equivalent if they produce the same output streams when reading the same input streams. Along this line, several formal models with subtle differences have been defined to decouple timing from data while preserving the equivalence of streams [8], [33]. We will refer to these models as elastic timing models.

Figure 5 depicts four equivalent streams. They transfer the same data in the same order but at different time instants. The first stream, s_1, could correspond to a rigid synchronous timing model. Streams s_2 and s_3 could represent two equivalent streams regulated by a synchronous clock in which data transfers occur at different cycles and some of the cycles, depicted as shadowed boxes, contain invalid data (bubbles). Finally, s_4 could represent an asynchronous stream in which no global clock is regulating the data transfers.

At this point, an important observation can be made. When using elastic timing models, the latency of the system can be modified and, thus, long combinational paths can be broken by new sequential elements. This opens the door to a new avenue of exploration techniques that can synthesize circuits with different area/performance/power trade-offs, in a similar way as different pipelines can be proposed for a microprocessor without modifying the Instruction Set Architecture (ISA).

B. Introducing elasticity in hardware

The KPN model assumes unbounded channels with non-blocking writes. For hardware implementations channels must be bounded and some strategy is required to determine the size of the FIFO. Two options can be considered here:

1) Calculating an upper bound of the size of the FIFOs considering all possible behaviors of the KPN.

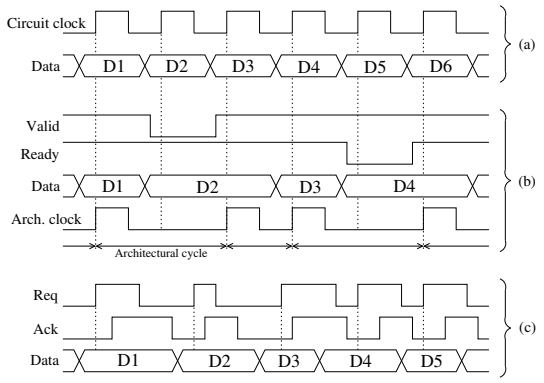


Fig. 6. Synchronization mechanisms: (a) synchronous rigid timing, (b) synchronous elastic timing, (c) asynchronous elastic timing.

2) Using blocking writes when FIFOs become full.

The first option is not always acceptable since boundedness imposes timing constraints on the environment. Moreover, the selected bounds also prevent a full flexibility in substituting components by other instances with different timing characteristics.

The second option is more appropriate for elastic timing but requires some handshaking mechanism to block the sender when the FIFO at the output channel becomes full. The size of the FIFOs is important in elastic systems since it directly determines the performance of the system and the absence of deadlocks. This aspect will be discussed in Section VII.

To put it simply, the transformation of a rigid system into elastic can be done by substituting the sequential elements (registers) by FIFOs controlled by handshake signals and following the firing semantics of KPNs.

Figure 6 shows timing diagrams for three signaling schemes corresponding to different timing models. The one in Fig. 6(a) corresponds to a rigid model in which data transfers occur at each cycle, i.e., no handshake signals are required. This scheme would produce streams similar to s_1 in Fig. 5.

The scheme in Fig. 6(b) represents a synchronous elastic channel with a Valid/Ready protocol. The *Valid* signal indicates when the sender has valid data, whereas the *Ready* signal indicates that the receiver is able to accept the data. Data transfers occur when *Valid* $\wedge$ *Ready*. This scheme is synchronous and the handshake signals are used to generate a gated version of the circuit clock (architectural clock) that triggers the data transfers. This scheme would produce streams similar to s_2 and s_3 in Fig. 5.

Finally, Fig.6(c) depicts a 4-phase asynchronous protocol. In this case, the *Request/Acknowledge* signals play a similar role as the *Valid/Ready* signals in the synchronous protocol. This scheme would produce streams similar to s_4 in Fig. 5. More details about the asynchronous protocols will be given in Section V.

We can observe that the three schemes are very similar and the only difference between rigid and elastic systems is the implementation of the communication channels. This similarity can be observed in Fig. 7, where three different implementations are depicted. The leftmost diagram (a) represents a rigid system with a global clock that triggers

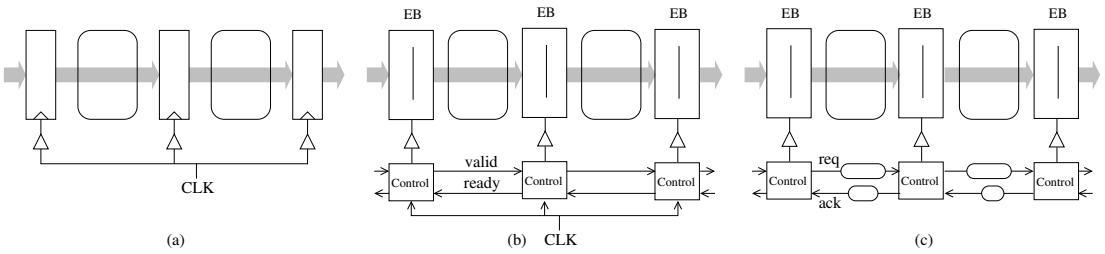


Fig. 7. The structure of the design: (a) synchronous rigid timing, (b) synchronous elastic timing, (c) asynchronous elastic timing (matched delays version).

the flip-flops between combinational regions. The diagram in the middle is synchronous elastic and has been obtained from the rigid system by substituting the flip-flops by elastic buffers (FIFOs). The controllers implement the handshake protocol and generate the architectural clock as a gated version of the circuit clock. Finally, the rightmost diagram is an asynchronous elastic system. The only difference with the synchronous elastic system is in the controllers that generate the architectural clock using the asynchronous handshake signals. The delays in the req/ack signals are designed to match the delays in the datapath in such a way that the clock edges satisfy the setup/hold timing constraints.

Up to this point, the paper has discussed the evolution of timing models to improve system performance. In the sequel, the paper will focus on elasticity and how it can be used to optimize a design up to the point of performing automatic pipelining.

V. Asynchronous Circuits

Asynchronous circuits have been present for many years in the academic community as an alternative to the synchronous [48]. They have also had a limited presence in industry. The goal of asynchronous design is to adapt the delays to the dynamic requirements of the computations and the environment of a circuit.

This section reviews some basic protocols that have been broadly used in asynchronous design with the goal of illustrating possible implementations for elastic timing models, such as the one shown in Fig. 6(c). We refer the reader to [48] for a more extensive review of the area.

One of the pioneering efforts in asynchronous systems was the Macromodules project [37] with the introduction of the *Delay-Insensitive* (DI) modules that can interact with the environment under the presence of arbitrary delays at the inputs and outputs of the module. Seitz introduced *self-timed systems* [45] with the assumption that the components of the system were working with local timing constraints (e.g. negligible wire delays).

In general, different forms of asynchronous circuits have been proposed, all of them aiming at avoiding a global synchronization at system level. As discussed in Section IV, the implementation of these type of timing schemes requires a set of handshake signals with a synchronization protocol.

Figure 8 illustrates the two most popular protocols using a pair of *request/acknowledge* signals. In the 4-phase protocol, a communication cycle involves four events and the handshake

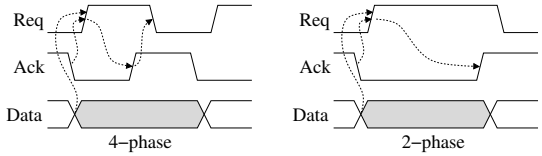


Fig. 8. Asynchronous 4-phase and 2-phase communication protocols.

signals return to zero at the end of each data transfer. In the 2-phase protocol, each cycle involves only two events (either rising or falling). Both protocols have been used extensively in asynchronous circuits, showing different advantages and disadvantages.

As shown in Fig. 7, the difference between synchronous and asynchronous timing models is mostly reduced to the implementation of the synchronization layer that generates the clock signal for the sequential elements of the circuit.

Figure 9 depicts one of the best known schemes for asynchronous pipelines, based on a 4-phase protocol implemented with Muller's C-elements [38]. The picture shows a linear pipeline with latches between regions of combinational logic (CL). The C-element at each latch guarantees that a latch becomes transparent when a new data is available at the input ($Req = 1$) and no data is stored in the following latch ($Ack = 0$). A FIFO (in the KPN sense) can be implemented as a sequence of latches with no logic in between. A similar scheme using a 2-phase protocol was proposed by Sutherland to implement the well-known *Micropipelines* [50].

The handshake protocol also requires certain timing constraints to guarantee that latches are neither missing nor overwriting data. For this reason it is necessary to guarantee a minimum separation of events that allows data to traverse combinational logic without having any setup/hold timing violation. Figure 9 shows a delay for every *Req* signal to properly synchronize the clock signals at each latch. There are different ways to implement this delay, either by a chain of logic gates that mimic the delay in the combinational logic (bundled-data approach) or by implementing the combinational logic with multiple rails (e.g., dual rail) and using explicit circuits to detect the completion of the computation.

The latter case can be pushed to the limit when no timing assumptions are required to guarantee the correct sequencing of events. This can be achieved by using delay-insensitive codes [52]. However, the robustness provided by these schemes comes with a high cost in area and power [49].

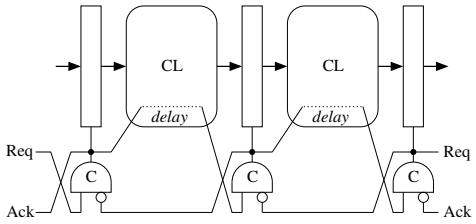


Fig. 9. Muller's pipeline

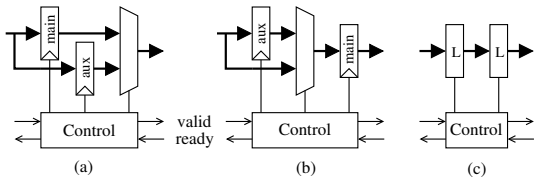


Fig. 10. Implementation of elastic buffers.

VI. Elastic Circuits

The worlds of synchronous and asynchronous designs converged from different starting approaches. On the one hand, [42] studied synchronous implementations of the asynchronous circuits. On the other hand, [10] proposed latency-insensitive systems. Both forms in essence implemented the idea of an asynchronous system quantized by the global clock reference as illustrated by the timing diagram in Figure 6(b) and the structural diagram in Figure 7(b).

Since global *stall* signals are typically unacceptable for large systems, a distributed control is required to handle the *back-pressure* generated when a receiving block is *not ready* to accept new data. Extra storage is necessary inside the elastic buffer[1] (EB) to accommodate new incoming data from the previous block to the stalled block, while preserving the previously received data that have not yet been processed. When the *ready* signal is asserted, the EB can deliver the stored data in FIFO order. A desirable property is that EBs do not introduce additional latency with regard to conventional synchronous registers in absence of back-pressure.

A. Design of elastic buffers and elastic pipelines

Figure 10 shows three implementation schemes for an EB that can hold two data items. Note that while this figure presents EBs capable of holding two data items, in general EBs are elastic FIFOs of arbitrary finite capacity.

Figures 10(a) and 10(b) show two possible implementations of an EB using edge-triggered registers. Other implementations are also possible, but in all of them a multiplexer is required to select data from one of the registers. Figure 10(c) shows the structure of a more efficient implementation using transparent latches [26], [16]. The use of flip-flop based registers though can sometimes be more convenient for timing analysis or for the implementation in FPGAs that do not provide support for latches.

The designs depicted in Figure 7 can be easily extended to netlists in which the elastic modules have multiple inputs and outputs connected to other elastic modules.

[1]Also called *relay station* in [10]

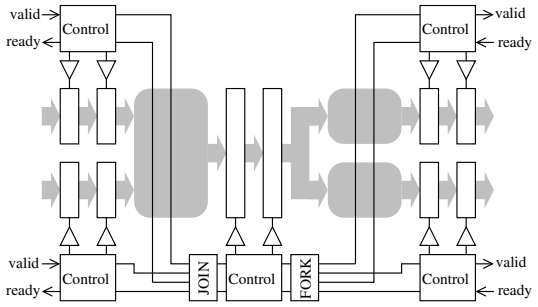


Fig. 11. Synchronous elastic module with multiple inputs and outputs.

An example of such extension is depicted in Fig. 11. The controller in the middle of the figure is synchronized with the neighboring controllers. The *Join* block combines the handshake signals of the input modules with the ones of the controller. Similarly, the *Fork* blocks combines the handshake signals for the output modules. Intuitively, the *Join* block implements the conjunction of *valid* signals, whereas the *Fork* block implements the disjunction of *stop* (not ready) signals. Note that the same scheme applies for an asynchronous circuit with *req/ack* handshake signals.

B. Performance analysis

The performance analysis of an elastic system is founded on the well-known theory of marked graphs [14], [43], which can be briefly summarized as follows.

Every directed cycle in the system has a number of tokens and bubbles stored in EBs and distributed in a number of stages. A data transfer in a cycle implies swapping the location of a token and a bubble, i.e., the token moves to an empty slot in an EB, whereas the previous location of the token becomes empty. The number of tokens and bubbles is an invariant for every cycle.

Every cycle C in the system has an associated throughput (tokens per cycle) that can be calculated as follows:

$$Th(C) = \frac{\min(T_C, B_C)}{S_C}$$

where T_C and B_C are the number of tokens and bubbles, respectively, and S_C is the number of stages in the cycle. Intuitively, the formula computes the average number of tokens per cycle that can be processed at every stage. However, given the duality between the move of tokens and bubbles, the performance may also be limited by the lack of space in the cycle (a token cannot move if there is no available space at the next stage). This is the reason for the *min* operator in the formula. In a strongly connected system with several cycles, the performance is determined by the most stringent cycle:

$$Th = \min_C Th(C).$$

Figure 12 depicts an example for performance analysis in which the shadowed ovals represent combinational logic and the boxes represent 2-slot EBs. The system has two simple cycles, C_1 and C_2, with the following characteristics:

$$T_1 = B_1 = S_1 = 4; \quad T_2 = 4, B_2 = 6, S_2 = 5.$$

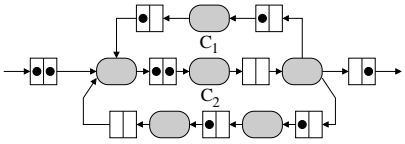


Fig. 12. Performance analysis of an elastic system.

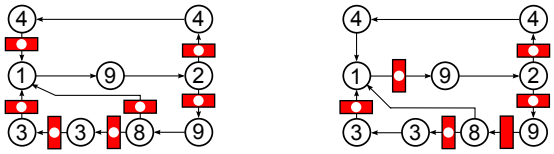


Fig. 13. Design after (a) retiming, (b) retiming and recycling.

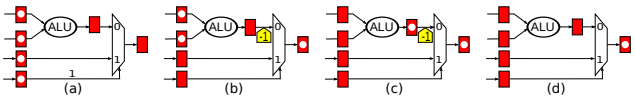


Fig. 14. Moving tokens and anti-tokens for early evaluation.

Therefore, $Th = \min(Th_1, Th_2) = \min(1, \frac{4}{5}) = \frac{4}{5}$, indicating that every computational unit will operate 4 out of 5 cycles, on average. It is interesting to note that the addition of one token in C_1 would have a negative impact on performance since $T_1 = 5$ and $B_1 = 3$, hence $Th = 3/4$.

Note that a cycle with zero tokens or zero bubbles has zero throughput (deadlock). Therefore, a necessary and sufficient condition for liveness is that every cycle has at least one token and one bubble.

Besides the global constraints, every EB requires a local performance constraint to sustain the maximum allowed throughput under the presence of back-pressure. In order to accommodate incoming tokens and locally propagate the handshake signals (valid/ready), every EB requires at least two slots. All the previous analysis can be extended to multi-cycle units or asynchronous systems with arbitrary forward/backward propagation delays [11].

VII. Optimization of Elastic Circuits

Section II-B describes some of the optimization transformations that can be applied for standard synchronous circuits. All of them are latency-preserving transformations in that they do not change the latencies of a computation as measured in clock cycles. All of these transformations can be applied to elastic designs.

Asynchronous and synchronous elastic designs, on the other hand, tolerate changes in latency. Such tolerance can be used to introduce novel correct-by-construction transformations enabling the exploration of new microarchitectural trade-offs [29], [21]. In some cases, the cycle time of the system can be reduced by increasing the latency of some operations, e.g., by introducing more pipeline stages. By properly balancing cycle time and throughput, the system with the optimal *effective cycle time* can be achieved. The effective cycle time is a performance metric similar to the time-per-instruction (TPI) in CPU design. It captures how much time is required to process one token of information - the smaller the better. For example, let us assume that the design in Fig. 12 has a cycle period of 500 ps. Given that $Th = \frac{4}{5}$, the effective cycle time is 625 ps ($500/Th$), indicating that a token is processed every 625 ps, on average.

One of the properties of elastic circuits is that they accept the insertion of empty buffers at arbitrary locations while preserving the behavior of the design. This can be done both for asynchronous circuits [39], [36] or synchronous elastic circuits [8], [33]. We often refer to these empty buffers as *bubbles*. The process of inserting bubbles into an asynchronous design was called *slack matching* [36], while for synchronous elastic - *recycling* [9].

The main reason for the insertion of *bubbles* is performance improvement. Figure 13(a) shows an example after an optimal retiming. The combinational nodes (shown as circles) are labeled with their delays. The boxes represent 2-slot elastic buffers. Those labeled with a dot contain one token (valid data) and one bubble, whereas the unlabeled ones contain two bubbles. The cycle time of this design is 17 time units.

Figure 13(b) shows an optimal configuration combining retiming and recycling. An empty elastic buffer is inserted into the bottom path. The cycle time has been reduced to 11 time units. The throughput is determined by the slowest cycle. The token/buffer ratios for each cycle are 1, 4/5 and 2/3. Therefore, the throughput is 2/3, and the average number of cycles to process a token is 3/2. This provides an *effective cycle time* of 16.5 time units ($16.5 = 11 \cdot 3/2$). It means that a new token is processed on average every 16.5 time units, which is an improvement compared to the 17 units of the optimally retimed design.

As explained in [11] another technique to optimize the elastic circuits is buffer sizing: increasing the capacity of EBs. Unlike recycling, buffer sizing can never degrade the cycle time of the design. Both techniques can be combined with retiming in an automated flow to optimize elastic designs. Next section will describe additional optimization techniques based on anti-tokens.

VIII. Anti-tokens

One of the important optimizations towards automatic pipelining was the introduction of early evaluation and anti-tokens. Early evaluation contributes to increase the performance of elastic systems, because the design does not have to stall waiting for irrelevant data. Early evaluation was proposed for asynchronous [5], [1] and synchronous designs [15].

A. Early evaluation

The execution model of conventional elastic systems is based on strict evaluation: a computation is initiated only when all input data are available. This requirement can be relaxed if *early evaluation* is used. For example, the behavior of a 2-input multiplexer can be modeled with the following statement:

$$z = \texttt{if } s \texttt{ then } a \texttt{ else } b.$$

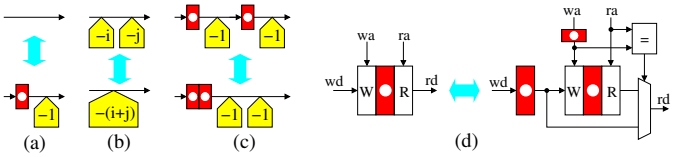


Fig. 15. Anti-token transformations: (a) insertion, (b) grouping, (c) retiming, (d) memory bypass.

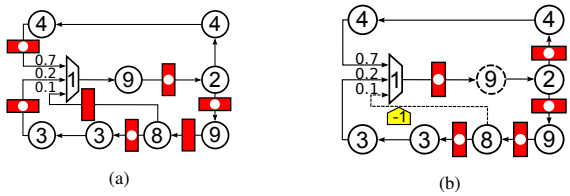


Fig. 16. Optimal configuration with (a) early evaluation and (b) anti-token insertion. Boxes represents 2-slot elastic buffers.

In case s is available and its value is *true*, there is no need to wait for b; the result can be delivered as soon as a arrives. Similarly when s is *false*.

When using early evaluation, the spurious enabling of functional units must be prevented if the late inputs arrive after the completion of the computation. One of the mechanisms to discard late inputs is the use of *negative tokens*, also called *anti-tokens*. Each time an early evaluation occurs, an anti-token is generated at every non-required input in such a way that it annihilates when it meets a positive token [15].

Figure 14(a) depicts a circuit with early evaluation. The circuit contains valid data in all registers with tokens. The 0-input of the multiplexer has no valid data, however the control signal indicates that the 1-input must be selected. In this situation, there is no need to wait for the "late" data produced by the ALU.

Figure 14(b) shows how the 1-input has been sent to the output of the multiplexer. Additionally, an *anti-token* (-1) has been sent backward to the 0-input. In Fig. 14(c), the result from the ALU arrives, and then in Fig. 14(d) the valid data and the anti-token cancel each other, thus disregarding the non-selected data.

Various implementations exist both in synchronous (e.g., [12]) and asynchronous (e.g., [1], [47]) circuits. Among the different implementations of anti-tokens, two main classes have been distinguished: *passive* anti-tokens, when they statically wait for the arrival of tokens, and *active* anti-tokens, when they move backward to meet tokens.

Active anti-tokens have the advantage of disabling data transfers proactively, potentially providing power savings (fewer computations performed). Passive anti-tokens are simpler to implement and have a lower area overhead.

B. Anti-token transformations

Anti-tokens can also be used to enable new retiming configurations beyond the ones presented in Section II-B. An empty EB is equivalent to an EB with one token of information followed by an anti-token injector with one anti-token, as shown in Fig. 15(a). An anti-token injector can be implemented with a simple control that contains an up-down counter placed on the communication channel (without any latency penalty). When a token flows through a non-empty anti-token injector, the token and the anti-token cancel each other, updating the counter.

Anti-tokens can be retimed (as in Fig. 15(c)) and grouped (as in Fig. 15(b)). However, anti-token counters cannot be retimed through computational units that have state or memory. Such state can be represented in the microarchitectural graph as a self-loop of the node, which will effectively disable retiming and propagation of anti-tokens.

C. Re-designing for average performance

The performance of a system with early evaluation is no longer determined by the slowest cycle, since average-case performance is achieved instead of worst-case. Early evaluation allows to design for the typical case in terms of data variability. Data that is not selected often can be safely delayed by a few clock cycles without a significant impact on the throughput. By using this technique, it is possible to achieve a cycle time reduction with small throughput degradation, thus improving the effective cycle time. Relaxing the timing constraints on parts of the design that are not used very often can also result in power and area savings.

Figure 16(b) shows a design similar to Fig. 13(b), with an early evaluation multiplexer selecting between three branches. In this example, anti-token insertion has been applied to the dashed channel. Then, the new EB can be retimed backwards across the node with delay 8, and the bubble between nodes 8 and 9 can be removed. This new configuration has a cycle time of 11 units, but the estimated throughput using an ILP model [7] is higher, 0.918, since there is only one cycle with a bubble[2] compared to Fig. 16(a), where two of the three cycles have bubbles. The resulting effective cycle time is 11.98 units. This configuration can only be achieved by using the anti-token insertion transformation.

There is no known efficient exact method to compute the throughput of a system with early evaluation. An upper bound method using linear programming is presented in [27]. Each input must be assigned a probability so that the performance can be analyzed. Such probabilities should be obtained by running a typical application on the system, and then counting how often each input is selected.

D. Memory bypass

One of the key strategies for boosting performance in computer architecture is to parallelize instruction execution when data and control dependencies do not prevent it. A common technique for that is to bypass data that have not been yet committed to memory or to the register file. These schemes introduce multiplexers to forward data to the requesting units.

Figure 15(d) depicts a transformation to introduce a memory (or register file) bypass [29]. The memory block is represented

[2]the cycle corresponds to the one with the anti-token (-1), as the sum of a token and an anti-token is equivalent to a bubble.

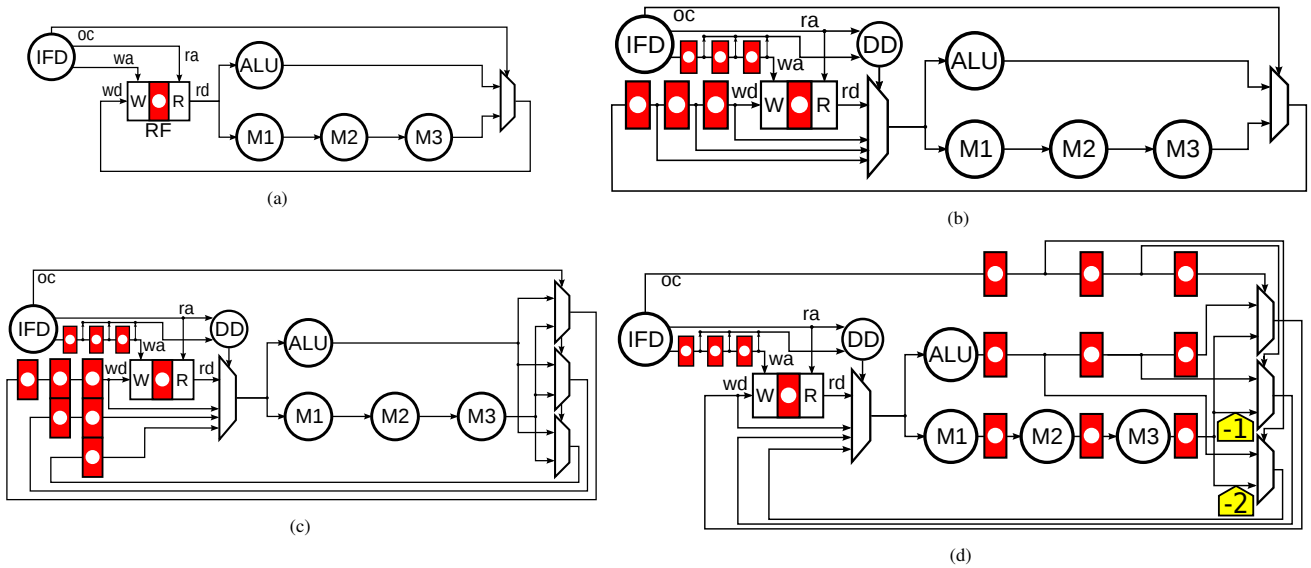


Fig. 17. (a) Graph model of a simple design, (b) After 3 bypasses, (c) Duplicate mux, enable forwarding, (d) Final pipeline after transformations

by the write logic (W), the read logic (R) and a buffer containing the information. A memory can be modeled as a large register that is read/written at every cycle, even though the read/write operations only affect a small subset of bits.

The transformation introduces an EB that delays the commitment of the write data (wd). Additionally, a register to delay the write address (wa) is required. The output multiplexer can select between the data coming from memory or the one coming from the non-committed data. The selection is done depending on the coincidence of the read address (ra) and the previous write address.

The relevance of this transformation is twofold. On one hand, a new EB is introduced that can be retimed backwards searching for more pipelining opportunities that can reduce the cycle time. On the other hand, the output multiplexer can perform early evaluation and forward data one cycle earlier when no data dependencies exist between the read and write requests. Interestingly, this transformation can be applied iteratively, creating a several forwarding levels.

Memory bypass is one of the key enablers of architectural exploration. By applying a different number of bypass transformations in the architecture, various configurations trading-off area and performance can be generated and compared.

IX. Automatic Pipelining

By using the techniques presented in previous sections, it is possible to automatically pipeline a design starting from a functional specification graph. Pipelining is achieved by by applying a sequence of correct-by-construction structural transformations that generate an elastic distributed control in which every block only controls the input/output EBs and local computation blocks. The generated pipelines can handle dynamic data dependencies and do not rely on a static global schedule.

Bypassing memory elements using early-evaluation is essential for automatic pipelining, since they introduce new EBs that can be retimed backwards. Finally, the system can be pipelined by retiming the EBs inserted with the bypasses and using other transformations such as recycling or anti-token insertion.

A. Example: Pipelining a Reduced Instruction Set

Figure 17(a) shows the specification of a simple design. The register file *RF* is the only state holding block. *IFD* fetches instructions and decodes the opcode and register addresses. *ADD* and *M* are arithmetic functions. The results are selected by the multiplexer for *RF* write-back. *M* has been divided into three stages. The potential boundaries for breaking up logic to allow pipelining is a design decision that is typically determined a priori, regardless whether pipelining will be applied or not.

In Fig. 17(b), three memory bypasses have been added to *RF* for building a bypass network. The node *DD* receives all previous write addresses and the current read address in order to detect any dependencies and determine which of the inputs of the bypass multiplexer must be selected.

The right-most multiplexer and the bypass EBs must be duplicated to feed each bypass path independently, enabling new forwarding paths, as shown in Fig. 17(c). Once the forwarding paths have been created, the design can be pipelined by applying retiming and anti-token insertion, as shown in Fig. 17(d). The final elastic pipeline is optimal in the sense that its distributed elastic controller inserts the minimum number of stalls. Furthermore the pipeline structure is not redundant since there are no duplicated nodes. Therefore, this is as good as a manually designed pipeline.

Fast instructions that require few cycles to compute, like *ADD* in this example, use the bypass network to forward data avoiding extra stalls, while long instructions use the bypass network as a stall structure that handles data hazards. In this example, the only possible stalls occur when the paths with anti-token counters are selected by the early evaluation multiplexers. This situation corresponds to a read after write

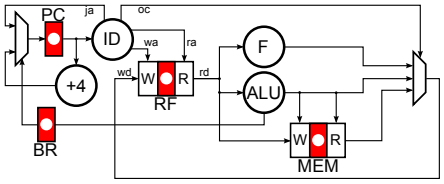


Fig. 18. DLX initial graph

TABLE I
DELAY, AREA AND LATENCY NUMBERS FOR DLX EXAMPLE

Block	Delay	Area	Lat.	Block	Delay	Area	Lat.
mux2	1.5	1.5	1	EB	3.15	4.5	1
ID	6.0	72	1	+4	3.75	24	1
ALU	13.0	1600	1	F	80.0	8000	1
RF W	6	6000	1	RF R	11	-	1
MEM W	-	-	1	MEM R	-	-	10

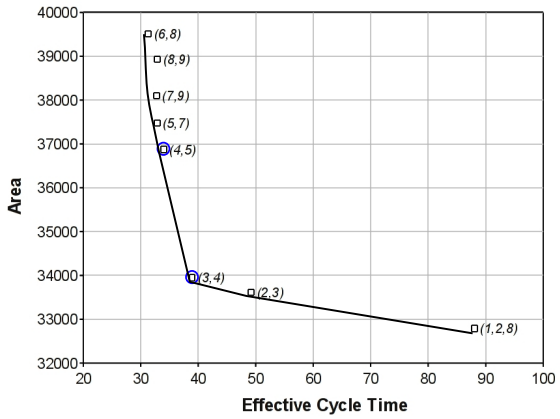


Fig. 19. Effective cycle time and area of the best pipelined design for different depths of F. (x,y) and (x,y,z) tuples represent the depth of F, the number of bypasses applied to RF and to MEM ($z = 9$ if omitted)

(RAW) dependency involving a result computed by *M*, which needs three cycles to complete.

B. Microarchitectural exploration

As the graph specification grows, the number of possible pipelining configurations explodes exponentially, and manual exploration becomes complicated and error-prone. The retiming and recycling optimizations, including anti-token insertion and retiming, can be unified as a mixed integer linear programming problem (MILP) [7], and solved using linear programming tools. Therefore, automatic exploration of pipelines can be achieved by trying different combinations of bypasses added to each of the memory elements of the design, and then automatically pipelining each exploration point [20]. Since the retiming and recycling method can only compute an upper bound of the throughput instead of the exact throughput, the most promising designs should be simulated at the end of the exploration to identify the best one, or to study a possible trade-off between performance and area or power.

Let us illustrate this exploration method on a simple microarchitecture similar to a DLX, shown in Fig. 18 before pipelining. In this microarchitecture, there is no pipelining and every instruction is executed in one cycle with a long cycle time. The execution unit has an integer ALU and a long operation F. The instruction decoder ID produces the opcode, oc, that goes to the write-back multiplexer and a jump address, ja, which depends on the previous ALU operation, stored in register BR. Table I shows approximate normalized area and delays of the functional blocks. These parameters have been obtained by synthesizing some of the blocks in a 65nm technology library (ALU, RF, mux2, EB and +4). The rest of the values have been estimated.

F is considered to be a floating point unit which may be divided into several blocks for pipelining ($F_1, \ldots F_i$). The memory has a read latency of L_{MEM} cycles, which is set to 10 in Table I. This is a typical value for the read latency of an on-chip cache memory.

Figure 19 shows the effective cycle time and area of the best Pareto points found for different partitions of F. Using a set of realistic assumptions for the probability of each instruction and the probability of data hazards, a pipeline was generated automatically for each possible partition of F, collecting the effective cycle time of the resulting pipeline and its area. It also shows how many bypasses were added to the register file and the memory to introduce instruction parallelism. This figure illustrates how more pipelining implies better performance at the expense of more area. While the best performance is achieved with 6 stages, shorter pipelines (such as the ones circled in the figure) are simpler and can offer attractive solutions.

Figure 20 shows one of the best design points, with F partitioned into three blocks. 3 bypasses have been applied to RF and then EBs have been retimed to pipeline F. Note that an extra bubble has been inserted at the output of F_3: the reduction in the throughput due to this bubble is compensated by a larger improvement in cycle time. Without this bubble the critical path would include the delay of the multiplexers after F_3. This kind of decisions are driven by the probabilities at the multiplexers, and derived automatically by the MILP model.

The bypasses in the memory MEM implicitly create a *load-store buffer* to hide memory latency, as shown in Fig. 20. Such structure can be substituted by a more efficient implementation: an *associative memory*. The exploration algorithm can detect the need for such load-store buffers and calculate their optimal size.

Figure 21 illustrates how a simple program executes in this example. Fig. 21(a) shows the trace of a set of 5 instructions executed in the original DLX design from Fig. 18. In this non-pipelined design, the cycle time is 107.65 units (using the data in Table I). Each instruction takes one cycle to complete and the program runs in 538.25 time units.

Fig. 21(b) shows the trace corresponding to the execution of the same program in the pipelined version of the DLX shown in Fig. 20. In this case, since there are three stages in F, the cycle time is reduced down to 30 units. However, it needs 11 cycles to complete, so the total execution time is 330 units of time. The ALU stage in the trace corresponds to the execution of the ALU function in Fig. 20, while ALU2 is the empty stage starting with the EB right next to the ALU unit.

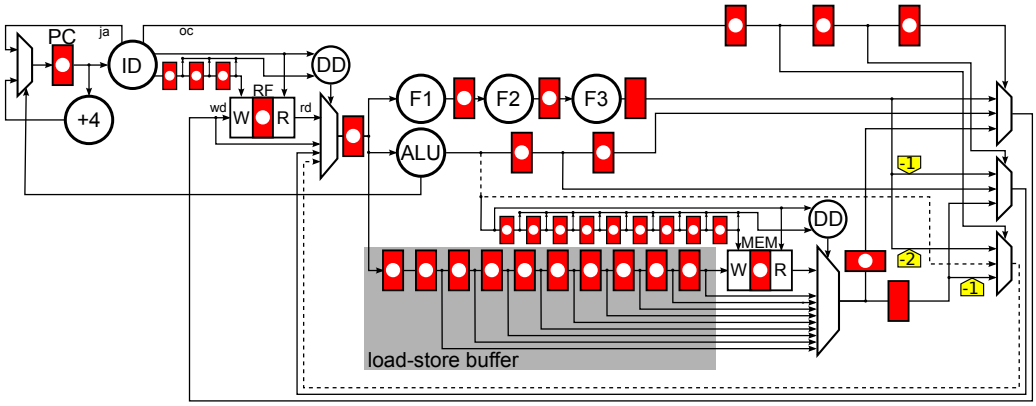
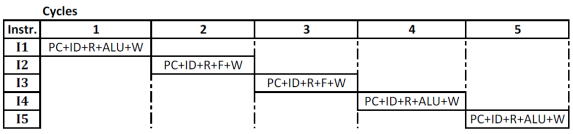
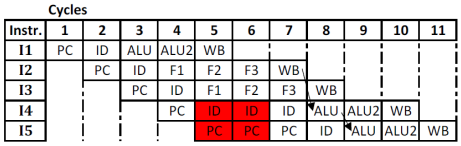


Fig. 20. Pipelined DLX (F split into 3 blocks. RF has 3 bypasses and M 9)



Instr.	Cycles 1	2	3	4	5
I1	PC+ID+R+ALU+W				
I2		PC+ID+R+F+W			
I3			PC+ID+R+F+W		
I4				PC+ID+R+ALU+W	
I5					PC+ID+R+ALU+W

(a)



Instr.	Cycles 1	2	3	4	5	6	7	8	9	10	11
I1	PC	ID	ALU	ALU2	WB						
I2		PC	ID	F1	F2	F3	WB				
I3			PC	ID	F1	F2	F3	WB			
I4				PC	ID	ID	ID	ALU	ALU2	WB	
I5					PC	PC	PC	ID	ALU	ALU2	WB

(b)

Fig. 21. Execution of 5 instructions for the unpipelined and pipelined designs of the DLX example in this section, (a) in the unpipelined DLX (b) in the pipelined DLX. The 5 instructions are I1=ALU, I2=F, I3=F, I4=ALU, I5=ALU. I4 needs to read the result from I2, and I5 needs to read the result from I4.

There are two data dependencies during this execution run: I4 reads the result from I2, and I5 from I4. The data dependencies are drawn with an arrow in the trace. For the first dependency, the execution of I4 and I5 is stalled during two cycles as the data hazard cannot be resolved until F1-F2-F3 is completed. Therefore, two bubbles are introduced in the pipeline (shaded boxes in Fig. 21). For the second dependency, there is no need to stall, since the ALU unit completes within one cycle, and there is one forwarding path that can be used to bypass the result to the next instruction. This forwarding path is represented by a dashed line in Fig. 20.

As explained above, this example shows how the bubble added at the end of the F pipeline is useful to further reduce the cycle time. If instructions using F occur with low probability, this bubble will be automatically inserted by the exploration algorithm, thus reducing cycle time with a small throughput penalty. If F is executed with high probability, the bubble will not be inserted because the gain in cycle time will not compensate the throughput penalty.

X. Conclusions

After the classical optimizations in logic synthesis, automatic pipelining is the next step in design automation that contributes to improve the power/performance trade-offs in system design.

The paper has presented the historical evolution of design automation that has enabled novel optimizations for pipelining. Although the proposed transformations can improve performance by introducing handshake-based elastic timing, the behavior still preserves the so-called in-order execution, i.e. the generated traces transfer information in the same order as in the original non-elastic system.

Several problems still remain opened. Equivalence checking is one of the main challenges for transformations that modify timing beyond the natural boundaries of RTL state signals. Even today, retiming has a very limited use due to the limited scalability of the sequential equivalence algorithms. Some initial efforts have already been published [53], but there is still a long way to go until commercial verification tools can adopt this technology.

Automation for out-of-order execution is another challenge that deserves further investigation. While this paradigm is highly exploited in advanced CPUs, there is still little progress in automatic synthesis.

References

[1] M. Ampalam and M. Singh. Counterflow pipelining: Architectural support for preemption in asynchronous systems using anti-tokens. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 611–618, 2006.

[2] ARM Limited. *AMBA™ Specification (Rev 2.0)*, 1999.

[3] R. Brayton, G. Hachtel, and A. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, Feb 1990.

[4] R. K. Brayton, A. L. Sangiovanni-Vincentelli, C. T. McMullen, and G. D. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Norwell, MA, USA, 1984.

[5] C. Brej and J. Garside. Early output logic using anti-tokens. In *Int. Workshop on Logic Synthesis*, pages 302–309, May 2003.

[6] D. Buchfuhrer and C. Umans. The complexity of Boolean formula minimization. *Journal of Computer Science and System Sciences*, 77(1):142–153, Jan. 2011.

[7] D. Bufistov et al. Retiming and recycling for elastic systems with early evaluation. In *Proc. ACM/IEEE Design Automation Conference*, pages 288–291, July 2009.

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

[9] L. Carloni and A. Sangiovanni-Vincentelli. Combining retiming and recycling to optimize the performance of synchronous circuits. In *16th Symp. on Integrated Circuits and System Design (SBCCI)*, pages 47–52, Sept. 2003.

[10] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sagiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 309–315, Nov. 1999.

[11] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin. Elastic circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(10):1437–1455, Oct 2009.

[12] M. Casu and L. Macchiarulo. Adaptive Latency-Insensitive Protocols. *IEEE Design & Test of Computers*, 24(5):442–452, 2007.

[13] S. Chatterjee. *On algorithms for technology mapping*. PhD thesis, University of California at Berkeley, Berkeley, CA, 2007.

[14] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.

[15] J. Cortadella and M. Kishinevsky. Synchronous elastic circuits with early evaluation and token counterflow. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 416–419, June 2007.

[16] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 657–662, July 2006.

[17] P. Coussy and A. Morawiec, editors. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

[18] E. J. McCluskey, Jr. Minimization of Boolean Functions. *Bell System Technical Journal*, 35(6):1417–1444, Nov. 1956.

[19] J. P. Fishburn. Clock skew optimization. *IEEE Transactons on Computers*, 39(7):945–951, July 1990.

[20] M. Galceran-Oms, J. Cortadella, M. Kishinevsky, and D. Bufistov. Automatic microarchitectural pipelining. In *Design, Automation and Test in Europe*, pages 961–964, Apr. 2010.

[21] M. Galceran-Oms, A. Gotmanov, J. Cortadella, and M. Kishinevsky. Microarchitectural transformations using elasticity. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 7(4):18, 2011.

[22] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: a high-level synthesis framework for applying parallelizing compiler transformations. In *Proc. International Conference on VLSI Design*, pages 461–466, Jan. 2003.

[23] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau. Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 9(4):441–470, Oct. 2004.

[24] C.-T. Hwang, Y.-C. Hsu, and Y.-L. Lin. PLS: a scheduler for pipeline synthesis. *IEEE Transactions on Computer-Aided Design*, 12(9):1279–1286, Sept. 1993.

[25] *IEEE Standard for A Versatile Backplane Bus: VMEbus*, 1987. IEEE Std 1014™-1987 (R2008).

[26] H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers. Synchronous interlocked pipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 3–12, Apr. 2002.

[27] J. Júlvez, J. Cortadella, and M. Kishinevsky. Performance analysis of concurrent systems with early evaluation. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, Nov. 2006.

[28] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[29] T. Kam, M. Kishinevsky, J. Cortadella, and M. Galceran-Oms. Correct-by-construction microarchitectural pipelining. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 434–441, 2008.

[30] K. Keutzer. DAGON: Technology binding and local optimization by DAG matching. In *Design Automation Conference*, pages 341–347, June 1987.

[31] A. Kondratyev, L. Lavagno, M. Meyer, and Y. Watanabe. Realistic performance-constrained pipelining in high-level synthesis. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1–6, Mar. 2011.

[32] D. Kroening and W. Paul. Automated pipeline design. In *Proc. ACM/IEEE Design Automation Conference*, pages 810–815, June 2001.

[33] S. Krstic, J. Cortadella, M. Kishinevsky, and J. O'Leary. Synchronous elastic networks. In *FMCAD*, pages 19–30. IEEE Computer Society, 2006.

[34] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[35] N. Maheshwari and S. S. Sapatnekar. An improved algorithm for minimum-area retiming. In *ACM/IEEE Design Automation Conference*, pages 2–7, 1997.

[36] R. Manohar and A. J. Martin. Slack elasticity in concurrent computing. In J. Jeuring, editor, *Proc. 4th International Conference on the Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 272–285, 1998.

[37] C. E. Molnar, T.-P. Fang, and F. U. Rosenberger. Synthesis of delay-insensitive modules. In H. Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.

[38] D. E. Muller. Asynchronous logics and application to information processing. In *Symposium on the Application of Switching Theory to Space Technology*, pages 289–297. Stanford University Press, 1962.

[39] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, Apr. 1959.

[40] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

[41] E. Nurvitadhi, J. Hoe, T. Kam, and S.-L. L. Lu. Automatic Pipelining from Transactional Datapath Specifications. *IEEE Transactions on Computer-Aided Design*, 30(3):441–454, Mar. 2011.

[42] J. O'Leary and G. Brown. Synchronous emulation of asynchronous circuits. *IEEE Transactions on Computer-Aided Design*, 16(2):205–209, Feb. 1997.

[43] C. V. Ramamoorthy and G. S. Ho. Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Trans. Software Eng.*, 6(5):440–449, 1980.

[44] M. Riedel and J. Bruck. Cyclic Boolean Circuits. *Discrete Applied Mathematics*, 160(3):1877–1900, 2012.

[45] C. L. Seitz. System timing. In C. A. Mead and L. A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[46] N. Shenoy and R. Rudell. Efficient implementation of retiming. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 226–233, 1994.

[47] D. Sokolov, I. Poliakov, and A. Yakovlev. Analysis of static data flow structures. *Fundamenta Informaticae*, 88(4):581–610, 2008.

[48] J. Sparsø and S. Furber, editors. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.

[49] J. Sparsø and J. Staunstrup. Delay-insensitive multi-ring structures. *Integration, the VLSI journal*, 15(3):313–340, Oct. 1993.

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

[51] C. Umans, T. Villa, and A. Sangiovanni-Vincentelli. Complexity of two-level logic minimization. *IEEE Transactions on Computer-Aided Design*, 25(7):1230–1246, July 2006.

[52] T. Verhoeff. Delay-insensitive codes—an overview. *Distributed Computing*, 3(1):1–8, 1988.

[53] V. Wijayasekara and S. Srinivasan. Equivalence checking for synchronous elastic circuits. In *IEEE/ACM Int. Conf. on Formal Methods and Models for Codesign (MEMOCODE)*, pages 109–118, Oct. 2013.