UNIVERSITAT POLITÈCNICA DE CATALUNYA
DEPARTAMENT DE LLENGUATGES I SISTEMES INFORMÀTICS

Automatic Pipelining of Elastic Systems

Marc Galceran-Oms

Advisors: Jordi Cortadella and Mike Kishinevsky

June 2011

Preface

The structure of the pipeline is one of the key decisions made during the early design stages of a digital circuit. However, it is often decided based on intuition and the experience of the architects, mainly because of the significant computational costs of design space exploration and the lack of analytical optimization methods capable of finding a good pipeline automatically.

This work presents an optimization method that automatically explores microarchitectures in order to find and evaluate different pipeline options. Exploration is enhanced by the capabilities of Synchronous Elastic Systems and it is accomplished by applying provably correct-by-construction transformations on the microarchitecture. Elastic systems can tolerate latency changes in computations and communications. This elasticity enables new microarchitectural trade-offs aiming at average-case optimization (in terms of data variability) rather than worst case.

The first contribution of this work is to present a list of correct-by-construction transformations, such as empty buffer insertion, early evaluation, variable-latency computations or sharing of functional units. These transformations are local, i.e., they are applied to a small part of the microarchitecture, and it is possible to pipeline a microarchitecture by applying a sequence of such transformations. Some of them can only be applied to elastic systems, since they modify the latency of the communications and computations.

The second contribution uses the collection of transformations to introduce speculative execution in an elastic system. This solution can be used to implement some well-known architectural features like branch prediction or error detection and correction schemes.

The third contribution uses elastic transformations to automatically pipeline a microarchitectural description of a system or a partially pipelined design. A novel framework for microarchitectural exploration is introduced, showing that the optimal pipelining of a circuit can be automatically obtained. The optimization is guided by the expected workload of the system.

Finally, a method to analyze the performance of elastic systems and obtain a lower bound of the throughput in presence of early evaluation and variable latency is presented. There were already existing methods to obtain an upper bound of the throughput using linear programming, but there is no known tractable method to analyze the exact throughput of an elastic system with early evaluation.

Acknowledgments

This thesis would not have been possible without the support of many people.

I would like to acknowledge my advisors Jordi Cortadella and Mike Kishinevsky for all the guidance, support and motivation over the last four years. I am thankful for the opportunities they have provided and for all I have learned from them.

During these years, I have had several discussions with people that have helped me to better understand electronic design automation in general and elastic circuits in particular. My gratitude to Timothy Kam, Alexander Gotmanov, Alexey Pospelov, Satrajit Chatterjee, Dmitry Bufistov, Jorge Júlvez, Josep Carmona, Luciano Lavagno, and also to the Strategic CAD Labs group in Intel Corporation.

I am indebted to Sergi Oliva, Dmitry Bufistov, Nikita Nikitin, Jorge Muñoz, Adrià Gascón, Josep Lluís Berral, Dani Alonso, Miquel Camprodon, Manel Rello and Juan Antonio Farré for creating a great environment in the office. They have enriched these years with hours of discussions in topics as diverse as latex and programming tips, research ideas, music, sports, politics, and a software to predict soccer results (with little success). I must also thank Ferran Martorell, Jonàs Casanova, Juan Antonio Moya, Luca Necchi and Carlos Macián for making the final writing of this thesis an exciting period.

This thesis has been supported by a scholarship from Generalitat de Catalunya (FI), by a grant from Intel Corporation and the research project FORMALISM CICYT TIN2007-66523.

Finally, thanks to all my family and friends for always being there for me. I also want to express my gratefulness to Silvia for all the love, patience and encouragement.

List of Figures

1.1 Synchronous circuit abstraction 2
1.2 Basic RISC pipeline 3
1.3 Asynchronous pipeline 5
1.4 Synchronous and elastic traces 7

2.1 Petri net before and after firing transition t. 15
2.2 Behavior of a design with two internal registers, a and b. At each cycle, a different value is stored at the registers. In the elastic behavior, the symbol '*' denotes cycles with non-valid data (bubbles). 20
2.3 The SELF protocol. 22
2.4 Abstract model for an elastic FIFO. 23
2.5 Elastic buffer FSM and datapath connection 24
2.6 EB implementation 25
2.7 Skid-buffer implementation 26
2.8 Join and fork implementations 27
2.9 Early evaluation firing in transition t, an anti-token is added to the lower input place after firing. 28
2.10 Elastic channel with dual protocol. Valid$^+$ and Stop$^+$ propagate tokens forward, while Valid$^-$ and Stop$^-$ propagate anti-tokens backwards. 30
2.11 Abstract model for an elastic FIFO with anti-tokens. 31
2.12 Linear pipeline with dual elastic buffers. The anti-token flow is stopped at the beginning of the pipeline by setting S^-_{in} to 1. 32
2.13 Dual elastic controllers 34
2.14 Example of scheduling of elastic systems 36

2.15 Elastic system with 3 IP blocks that have been elasticized by adding an elastic wrapper around them. The control wires are marked with dotted lines. 38
2.16 (a) Simple synchronous pipeline, (b) Pipeline after converting flip-flops to EBs using a pair of latches, the register file is elasticized using a ghost latch, a bubble has been added after the instruction decoder (c) synchronous elastic system with controller; J means join, F fork and EJ early join; EB controllers are marked with a dot if they initially contain a token; paths from control to datapath and from datapath to control are dashed, (d) alternative way to elasticize the register file if the path from read address to read data is purely combinational. 41
2.17 (a) Simple elastic pipeline and (b) its elastic marked graph model, each pair of places represent an elastic buffer, the upper place is the forward place and the shadowed lower place is the backward place, which represents available capacity, (c) elastic buffer with joins and forks and (d) its elastic marked graph model, (e) equivalent elastic marked graph with join and fork structure explicitly modeled. . . 42
2.18 TMG model of a synchronous elastic system with two cycles. 44

3.1 Sequence of correct-by-construction transformations applied to a design. 50
3.2 Microarchitectural graphs 51
3.3 Bypass transformation 54
3.4 Retiming transformation 55
3.5 Example using retiming and recycling 56
3.6 Multiplexor retiming and bubble insertion 57
3.7 Capacity sizing in a channel. 58
3.8 Example of deadlock because of retiming 59
3.9 Example of retiming and recycling with early evaluation 61
3.10 Using little cares to optimize a design 62
3.11 Anti-token transformations 63
3.12 Passive vrs. active anti-tokens 65
3.13 Sharing transformation 68
3.14 (a) Datapath implementation of a shared module, (b) controller of the shared module. 68
3.15 Compositional correctness 70

4.1 Example of speculation in elastic systems: (a) non-speculative system, (b) system after bubble insertion, (c) system after Shannon decomposition, (d) final design with speculation. 74
4.2 Example where the outcome of the early-evaluation multiplexor is predicted using the proposed method. Each one of the tokens is labeled with a counter (i or $i-1$). The stage of EBs after the shared unit has a forward latency of L_f and a backward latency of L_b. 77
4.3 78
4.4 (a) Datapath and (b) control for an elastic buffer with no backward latency. 79
4.5 Speculation used for error correction and detection 81
4.6 Final speculative design corresponding to the original system in Fig. 4.5(a). 82
4.7 (a) Protocol to synchronize variable-latency units with elastic systems, (b) example of a generic variable-latency unit which takes 1 or 2 clock cycles. 83
4.8 Speculation used for variable latency. 84

5.1 Simple pipelining example 88
5.2 Simulation by case 90
5.3 Speed-up plot for the introduction example 90
5.4 Pipelining of a simple design with two instructions 91
5.5 Another possible pipeline for the graph in Fig. 5.4(a). 93
5.6 Example with three non-dominated Pareto-point configurations found by retiming and recycling. Their performance metrics are (a) $\tau = 1, \Theta = 0.5, \xi = 2$, (b) $\tau = 2, \Theta = 0.66, \xi = 3$, (c) $\tau = 3, \Theta = 1, \xi = 3$. 96
5.7 3 bypasses and retiming using anti-tokens 96
5.8 Example pipeline with forwarding paths 96
5.9 High level view of the exploration algorithm. 98
5.10 DLX initial graph. 102
5.11 Pipelined DLX graph (F divided into 3 blocks. RF has 3 bypasses and M 9). 104
5.12 Effective cycle time and area of pipelined designs 105
5.13 Throughput of the memory subsystem given different read latencies and bypass probability 107
5.14 Ring of pipelines. 108
5.15 Effective cycle time for different pipeline depth for $\gamma = 0.1$. 110

5.16 Effective cycle time for different pipeline depths for $k = 5$. 111
5.17 Flow for video decoding engine design experiment. 113
5.18 Area-performance trade-off of the original and the optimized decoder. 114

6.1 TGV example and firing example 118
6.2 2-period unfolding example 121
6.3 Unfolding example 126
6.4 Portion of an unfolding with reconvergent paths. 129
6.5 Graph representation of an MPEE expression 134
6.6 Graph representation of correlated expressions 136
6.7 Throughput of TGV computed using different correlation strategies 150
6.8 Run-time of the different performance evaluation versions in seconds for the graphs in Table 6.3. LB is the presented lower bound method, UB the linear programming upper bound method, and SIM is the simulation time. The Y-axis is in logscale. . . 154

7.1 Possible implementation for elastic multi-threading support. Grey boxes are shared units. 165

List of Tables

4.1 Example trace from Fig. 4.1(d) 76

5.1 Delay and area numbers for DLX example using NAND2 with FO3 as unit delay and unit area. 103
5.2 Parameter ranges in the experiments. 109
5.3 Number of bypasses performed in the pipelined design. 111
5.4 Run-time of the whole exploration method (including simulations) for different number of pipelines (k) and different depth ranges (D) in seconds. The minimum, maximum, mean, medium and quartiles are shown for each range. 112

6.1 Experimental results on random graphs with early-evaluation nodes and no variable-latency units 151
6.2 Experimental results on random graphs with variable-latency units and no early-evaluation nodes 152
6.3 Experimental results on random graphs with early-evaluation nodes and variable-latency units 153
6.4 Configurations found by the retiming and recycling method on graph G8. The first column shows the throughput found by simulation and the order according to this throughput. The second column shows the throughput found by the presented lower bound method and the corresponding order. Finally, the third column shows the throughput according to the upper bound method and the corresponding order. . 155
6.5 Number of swaps needed to produce a correct order. Confs shows how many RR configurations where created, LB and UB swaps show how many swaps were needed to achieve the correct configuration order, and “swap diff” shows the average throughput difference for the swapped configurations. 156

List of Algorithms

1 Bypass_One(G), exploration by adding one bypass at a time 100
2 Bypass_All(G), pipelining by adding one bypass per memory element at a time . . . 101
3 Top-level Exploration Algorithm. 101
4 Overview of the top level algorithm. 121
5 BuildMPEE, construct an MPEE expression from an unfolding. 133
6 EvaluateRecursive, algorithm that evaluates an MPEE expression. 138
7 A_exact, algorithm that evaluates an MPEE expression. 139
8 EvaluateUB, algorithm that computes an upper bound of the result of evaluating an MPEE expression. 141
9 A_prune, algorithm that computes an upper bound of all firing times by ignoring all correlations during evaluation. 141

Contents

1 Introduction **1**
1.1 Synchronous Circuits . . . 2
1.2 Pipelining . . . 3
1.3 Elasticity . . . 4
1.4 Why Synchronous Elastic Systems? . . . 8
1.5 Overview of the Contributions . . . 10
1.6 Organization of the Thesis . . . 13

2 State of the Art **15**
2.1 Preliminaries . . . 15
2.1.1 Petri Nets . . . 15
2.1.2 Marked Graphs . . . 16
2.1.3 Throughput of a TMG . . . 17
2.1.4 Linear Programming . . . 17
2.2 Synchronous Elastic Systems . . . 18
2.2.1 Definitions . . . 18
2.2.2 SELF Protocol . . . 21
2.2.3 Early Evaluation and Token Counterflow . . . 28
2.2.4 Verification . . . 35
2.2.5 Scheduling . . . 37
2.2.6 Granularity of elastic islands . . . 37
2.2.7 Synthesis Flow . . . 39
2.2.8 Modeling Elastic Systems with Marked Graphs . . . 41
2.2.9 Performance Analysis of Elastic Systems . . . 43

2.3 Automatic Pipelining . . . 46
2.4 Conclusions . . . 47

3 Correct-by-construction Transformations 49
3.1 Introduction . . . 49
3.2 Microarchitectural graphs . . . 51
3.3 Toolkit . . . 53
3.4 Latency-Preserving Transformations . . . 54
3.4.1 Bypass . . . 54
3.4.2 Retiming . . . 56
3.4.3 Multiplexor Retiming . . . 57
3.5 Recycling . . . 57
3.6 Buffer Capacities . . . 58
3.7 Early Evaluation . . . 60
3.7.1 Critical Cores and Little Cares . . . 61
3.8 Anti-token Insertion . . . 63
3.8.1 Passive or Active Anti-tokens . . . 64
3.9 Variable-latency Units . . . 66
3.10 Sharing of Functional Units . . . 67
3.11 Verification . . . 69
3.12 Conclusion . . . 71

4 Speculation 73
4.1 Introduction . . . 74
4.2 Speculative Execution . . . 75
4.3 EBs with Zero Backward Latency . . . 78
4.4 Design Examples . . . 80
4.4.1 Resilient Designs . . . 80
4.4.2 Variable-Latency Unit . . . 82
4.5 Conclusion . . . 84

5 Automatic Pipelining 87
5.1 Introduction . . . 88
5.2 Correct-by-construction Pipelining of a Simple Example . . . 89

5.3 Exploration Engine . . . 92
5.3.1 Retiming and Recycling . . . 94
5.3.2 Bypassing and Forwarding . . . 95
5.3.3 Two-phase Exploration . . . 98
5.3.4 Exploration Algorithm . . . 99
5.4 Pipelining Examples . . . 102
5.4.1 DLX pipeline . . . 103
5.4.2 Ring of pipelines . . . 108
5.5 Video Decoding Engine . . . 113
5.6 Conclusion . . . 115

6 Symbolic Performance Analysis 117
6.1 Introduction . . . 117
6.2 Problem Formulation . . . 123
6.2.1 Multi-guarded Marked Graph . . . 123
6.2.2 Unfoldings . . . 126
6.2.3 Single-Server Semantics . . . 127
6.2.4 Throughput . . . 127
6.2.5 Timing Simulation of a TGV . . . 128
6.3 Max-plus Algebra with Early Evaluation . . . 129
6.3.1 Definitions . . . 130
6.3.2 Representation of MPEE Expressions . . . 132
6.3.3 Lower and Upper Bounds of Expressions . . . 135
6.3.4 Exact Evaluation of MPEE Expressions . . . 135
6.3.5 Upper Bound Evaluation of MPEE Expressions . . . 139
6.3.6 Algorithm Complexity . . . 145
6.4 Number of Periods of the Unfolding . . . 146
6.4.1 Convergence of A_exact . . . 146
6.4.2 Convergence of the Upper Bound Method, A_prune . . . 148
6.4.3 Example . . . 149
6.5 Experimental Results . . . 151
6.5.1 Random Graphs . . . 151
6.5.2 Evaluating Relative Performance . . . 153

6.6 MPEE Simplification using SMT 155
6.7 Conclusion 157

7 Conclusions 161
7.1 Contributions 161
7.2 Further Research 163

Chapter 1

Introduction

CMOS process scaling, predicted by Moore's Law [114], enabled a steady increase in the complexity and the operating frequencies of integrated circuits. While the race on the operating frequencies has come to an end, technology shrinking continues, providing integration capacity of billions of transistors. The evolution of technologies enables the integration of complex systems in a single chip (System-on-Chip, SoC), and it has brought the popularization of multi-cores [18, 84, 124, 145].

As designs have become more and more complex, Electronic Design Automation (EDA, aka Computed Aided Design, CAD) has rapidly increased in importance. During the early years of the semiconductor industry, integrated circuits were designed by hand and the layout was also built manually. Incrementally, more and more stages of design were performed in an automatic or semi-automatic way. Place and route tools appeared, hardware description languages, verification tools, simulation tools, logic synthesis, etcetera. Today, EDA is present in all stages needed to build an integrated a circuit: design (from high level synthesis to physical synthesis), simulation, analysis, verification and manufacturing.

However, since technologies continue to shrink, the known problems become more challenging and new problems emerge. While synchronous circuits have been the dominant paradigm for circuit design through all these years, the asynchronous alternative has always been there claiming to provide a possible solution for the dark clouds that hover over the synchronous world.

Synchronous elastic systems have emerged in the last years as a middle ground solution between synchronous and asynchronous circuits. They can be designed and implemented using regular synchronous flows and tools, but they provide some of the advantages of asynchronous designs since they can tolerate variations in the latencies of the computations and the communica-

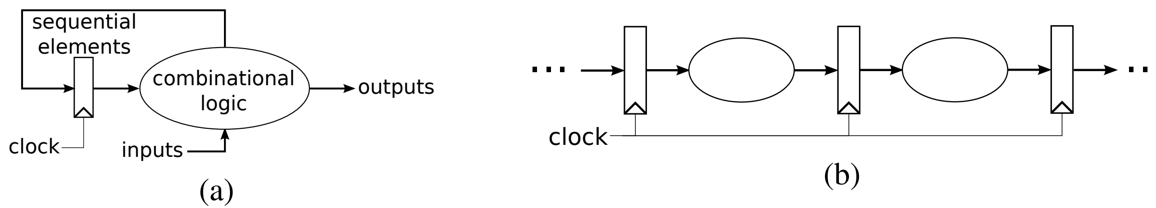


Figure 1.1: (a) Abstraction of a synchronous circuit, (b) Synchronous pipeline.

tions.

This tolerance can be used to separate performance critical parts from non-critical and optimize the former ones in isolation, while non-critical parts are removed from critical paths by adding latency to their computations. This is an example of the optimization opportunities that are enabled by elasticity. This thesis proposes a framework to leverage synchronous elastic systems in order to perform design space exploration of microarchitectures automatically. This chapter puts synchronous elastic systems into context and motivates the contributions of this thesis.

1.1 Synchronous Circuits

Figure 1.1(a) shows a very common abstraction for synchronous circuits. A synchronous circuit is formed by sequential elements (aka state signals), which are synchronized by an external clock, and the combinational logic, which performs the actual computation. The sequential elements load a new stable state each time a clock edge is delivered. The *clock* signal provides a time reference to the circuit.

Between two consecutive edges of the clock, the combinational elements compute the next stable state. Signals need some time to settle to their new state before a new clock edge can arrive. Otherwise, the results can be incorrect. The minimum time distance between two consecutive clock edges is the *cycle time* of the design, and it determines the maximum clock frequency at which the circuit can run. The slowest timing path determines the cycle time, and it is known as the *critical path* of the design.

The division of the computation in cycles delimited by a clock is very simple and intuitive. Furthermore, the separation between sequential elements that store the state and combinational elements that perform computations makes it easy to reason about synchronous systems. This simplicity is reflected in design, synthesis, timing analysis, testing and verification. Furthermore, most EDA tools target explicitly synchronous systems and most designers are only trained to use

Instr No.	**Pipeline Stage**						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Cycle	1	2	3	4	5	6	7

Figure 1.2: Basic RISC pipeline (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

them. Thus, the vast majority of circuits designed by the industry nowadays follow the synchronous paradigm.

1.2 Pipelining

In order to increase the throughput of a circuit, digital systems can be divided in stages connected forming a pipeline allowing the concurrent execution of multiple instructions or computations [140], as shown in Fig. 1.1(b). Each stage of the pipeline works on a part of an instruction while other stages work on other instructions in parallel. Each instruction enters at one end, progresses through the stages, and exits at the other end. Pipelining is a key implementation technique for the design of efficient circuits.

A pipeline is like an assembly line. Every step of an automobile assembly line works in parallel with the other steps, although on a different car. In a computer pipeline, each step (or stage) of the pipeline performs a different part of the computation. This way, the system clock can have a smaller cycle time, since the combinational logic between stages execute simpler computations compared to completing the whole computation in just one clock cycle, and hence it can run faster.

As an example, a basic RISC pipeline [75] typically has 5 stages, *Instruction Fetch, Instruction Decode, Execute, Memory access and Write Back*. Figure 1.2 shows an example execution for this basic pipeline. The first instruction is fetched on the first clock cycle. On the second one, the first instruction is decoded while the second one enters the pipeline and is fetched, and so on. On the sixth clock cycle, the first instruction has already been completed, and the second to the fifth instructions are being executed.

Ideally, the speed-up of a pipeline with 5 stages should be 5X, compared to the same un-

pipelined system. The same computation that had to be accomplished within one clock cycle can now be evenly divided in 5 parts, and hence each stage only needs 20% of the original cycle time. However, the ideal speed-up is typically not possible when the pipeline is deep enough. First, because it can be difficult to evenly balance the combinational logic among the stages, second, because of the overhead (setup time and clock skew) of the additional registers, and most important, because there may be dependencies between consecutive or close instructions.

While assembly code assumes that instructions are executed sequentially, pipelining invalidates this assumption. Data hazards appear when an instruction depends on the results of previous ones. These data dependencies can be solved by different techniques, such as forwarding and stalling. If the pipeline must be stalled, it is not possible to complete one instruction per cycle anymore. For example, consider the following instructions:

$$\begin{array}{l} add\ \$R_2,\ \$R_1,\ 1 \\ add\ \$R_4,\ \$R_2,\ \$R_3 \end{array}$$

The processor will add 1 to the contents of register R_1 and store it in R_2. Then, the execution of the second instruction will read registers R_2 and R_3, add their values and store the result in R_4. In a pipeline, the first instruction will be just one stage ahead of the second, so R_2 is not updated when the second instruction wants to read it. A way to solve this hazard is to stall the second instruction until the first one has completed. A second solution is to forward the result of the first instruction to the second one as soon as it is available, even if the first instruction has not completely finished. For example, an extra path can be added from the stage *Execute* to the stage *Decode* so that the result of the first instruction can be sent to the second one as soon as *Execute* knows the result for instruction *add*, avoiding stages *MEM* and *WB*.

In a microprocessor, the performance of a pipeline can be measured in instructions per cycle (IPC), i.e., the number of instructions completed on average per clock cycle. The inverse of the IPC, the CPI, or how many clock cycles are needed on average to complete an instruction, is also used. Finally, the *average time per instruction* can be found by multiplying the CPI by the cycle time of the circuit.

1.3 Elasticity

Since the early days of EDA, designers have been using optimization tools to improve the quality of the circuits (area, delay or power). For example, the techniques for two-level and multi-level

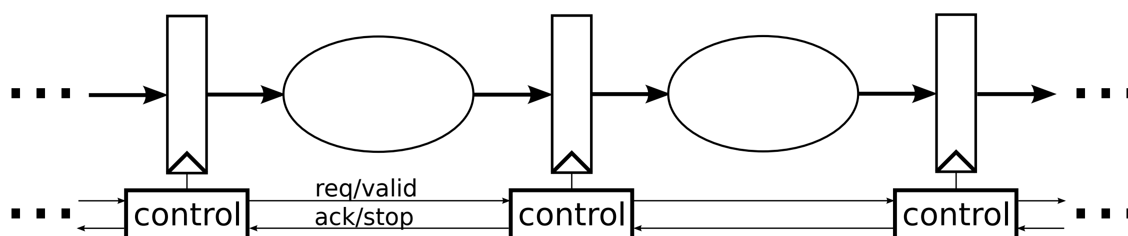


Figure 1.3: Asynchronous pipeline, each register is controlled by a local control which synchronizes with its neighbors.

combinational logic synthesis exploit the properties of Boolean algebra to transform gate netlists within the boundaries of the sequential elements of the circuit [58, 71]. With this approach, the behavior of the primary outputs and state signals is preserved, whereas the combinational parts are optimized.

A new generation of techniques enabled the crossing of the sequential boundaries and introduced new optimizations that can change the behavior of the state signals while preserving the behavior at the primary outputs. This field is known as *sequential logic synthesis* and includes transformations such as state encoding [8], redundant latch removal [14], retiming [98] and bypassing of memory elements [16].

All the previous transformations still preserve a *cycle-accurate* behavioral equivalence of the system: what is observable at the outputs of the system at the *i*-th clock cycle is independent from the optimizations performed on the system.

Maintaining the cycle accuracy imposes severe constraints on the type of optimizations that can be used in a circuit. For example, changing the structure of a pipeline or modifying the latency of a functional unit may not be applicable unless the global architecture of the system is transformed and adapted to the new timing requirements. This limitation is unacceptable for large systems that may be susceptible to late-stage re-design decisions to meet the timing specifications.

Elasticity has emerged as a new paradigm to overcome these limitations, enabling the design of systems that are tolerant to the dynamic changes of the computation and communication delays. The concept of elasticity has been widely used in asynchronous circuits. For example, the term *Micropipeline* was proposed by Sutherland [149] to denote event-driven elastic pipelines.

In contrast to synchronous circuits, asynchronous circuits are not governed by an external clock signal. Instead, each of its components is autonomous and it communicates with its neighbors in order to synchronize data flow, as shown in Fig. 1.3. This way delay variability can be tolerated. Typically, synchronization is performed by a handshake protocol where the sender indicates

whether it is ready to transmit new data and the receiver indicates whether it is ready. These handshake signals are often called *request* and *acknowledge*.

Different classes of asynchronous designs offer different advantages compared to synchronous designs. Asynchronous designs have the potential to provide low power consumption, because transistors only switch when a useful computation is performed. Asynchronous designs also have better modularity and composability, because timing constraints are more flexible and independent from module to module. In synchronous designs, it must be ensured that the circuit will meet the timing even in the worst-case conditions. Hence, the clock frequency is determined by the worst case. However, in asynchronous designs, the circuit speed adapts to the temperature and voltage conditions. This way, variability can be better tolerated. Furthermore, if the circuit is not operating under the worst-case conditions, the circuit can consume less power (or it can run faster).

Although it has various advantages, a very small part of the circuits are actually designed using this paradigm. One of the big disadvantages of asynchronous designs is that they typically require a completely different design flow. The tools to handle asynchronous circuits are less mature and most designers have more experience and expertise on designing synchronous circuits. Furthermore, the performance of the circuit is no longer determined by a fixed clock frequency, and it becomes more difficult to analyze what the actual performance is. There is an area overhead compared to synchronous designs because the handshake signals must be placed and routed along with the design. Finally, verification and testing of asynchronous designs is more challenging.

Elasticity can be discretized so that it can be implemented within a synchronous system, becoming a *Latency Insensitive System* [31, 32]. The handshake signals are then synchronized with the main clock of the system, i.e., each of the controllers shown in Fig. 1.3 implements a synchronous protocol instead of an asynchronous one. Many of the properties and optimization techniques of asynchronous designs are also applicable to latency-insensitive designs. In this context, the pair of handshake signals are typically called *valid* and *stop*, indicating respectively the validity of the data in the communication channel and the availability of the receiver to store the incoming data during each clock cycle. Different variants of synchronous elasticity have been proposed in the literature [53, 82, 127, 152].

The handshake controllers of elastic systems govern the flow of data along the datapath. Instead of implementing a monolithic stall controller that often resides on a critical path, the control is fully distributed alongside the datapath. Furthermore, each handshake controller produces a clock for the part of the datapath it manages. This way, each clock network is independent and becomes simpler to design.

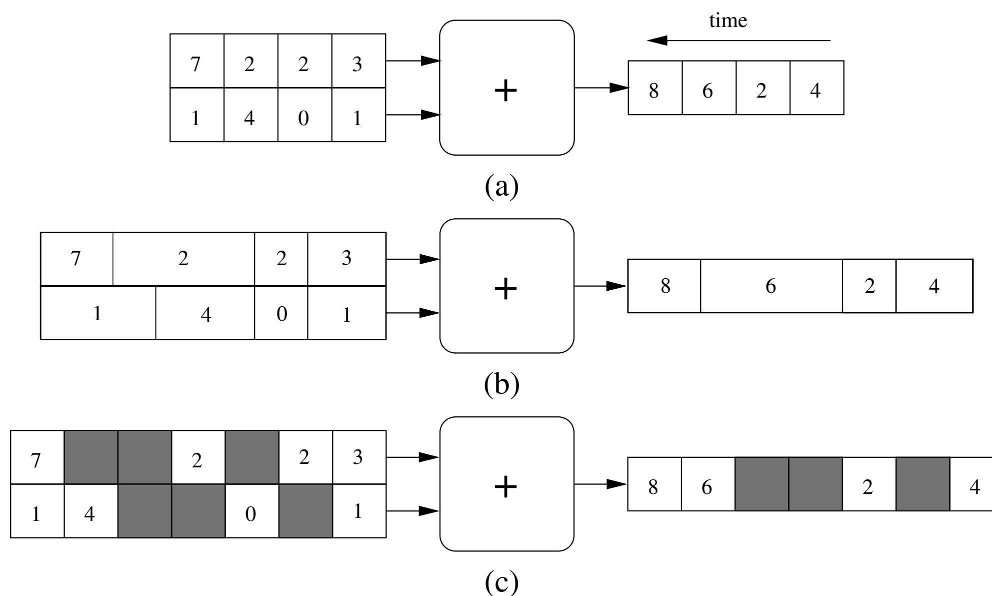


Figure 1.4: (a) Synchronous adder, (b) elastic adder, (c) synchronous elastic adder, shaded boxes represent idle cycles.

Latency-insensitive designs can tolerate any modification in the communication and computation latency. Thus, the functionality of the system only depends on the functionality of its components and not on their timing characteristics. The goal of this methodology is to guarantee that correct modules composed in a latency-insensitive framework will behave correctly.

Figure 1.4(a) shows an example of a conventional synchronous circuit performing additions. Each clock cycle, the circuit receives new inputs and produces the corresponding outputs. The environment is designed under the assumption that all operations will take one cycle.

With elasticity, the concept of behavioral equivalence is relaxed in a way that no cycle accuracy is required. Instead, the sequence of *valid* data is observed and preserved, as if one would connect FIFOs with non-deterministic delays at the inputs and outputs of the system. Figure 1.4(b) shows an example of an elastic system in an asynchronous environment, representing the same circuit. Inputs can arrive at any time. There is an underlying control that synchronizes them, the input that arrives earlier waits for the latest one. Once both inputs are available, the adder can produce a new output and send an acknowledge to the inputs so that they know they can send the next data item.

Figure 1.4(c) shows a synchronous elastic version of the same circuit, in which the empty cells represent non-valid data. In contrast to Fig. 1.4(b), time is discretized, and if data items are not available during one clock cycle, the module waits for the next clock cycle. It can be observed

that the traces of valid data for each signal are the same as in the non-elastic circuit. Although the timing is elastic, the causality relations between data items is preserved, e.g., the value $a + b$ is always generated after having received the corresponding values of a and b.

The main metric to measure the performance of an elastic system can still be the average time per instruction, as in a synchronous pipeline. In asynchronous systems the concept of cycle time is different because there is no clock. Thus, the time per instruction cannot be found just by multiplying the CPI by the cycle time, it is necessary to analyze how often actual computation occurs. Similarly, in synchronous elastic systems, the rate of cycles with valid computation must also be evaluated in order to determine the actual performance of the system.

1.4 Why Synchronous Elastic Systems?

Latency Insensitive (or synchronous elastic) systems offer some of the advantages of asynchronous systems, while they can be designed using synchronous CAD flows and tools. Elasticity may help solve or alleviate some of the current problems of circuit design. Let us review some of them and how elasticity may be a resource to mitigate these problems.

At the early 90s, the performance of a processor was considered to be determined by its clock rate. The advances in manufacturing, which allowed to double the number of transistors approximately every 18 months, allowed the frequency to double as well. The scaling of the clock frequencies was achieved by building deeper pipelines, providing more parallelism and hence a better IPC. Furthermore, a set of architectural techniques to reduce the stalls in the pipeline were introduced, such as out-of-order execution, threading, duplication of functional units and speculative execution.

By the mid 2000s, however, frequency scaling could not be sustained anymore. The improvement in frequency did not provide the same improvement in the actual performance of the circuit [3]: the level of parallelism could not be further improved within a single pipeline. Furthermore, supply voltage did not decrease at the same pace [129], and the power consumption rose considerably. Although there are some fundamental barriers [17], technology scaling still goes on nowadays. The extra space achieved by technology scaling is used to increase the performance of the circuits by adding more cores in a single chip working in parallel, going into Systems on Chip (SoC) and high-end chip multi-processors (CMPs). Besides, chips for mobile devices have emerged in the market. Such designs have tight power budgets in order to increase battery run-time. Thus, power has become an important design constraint in some circuits.

Elasticity can help to improve the overall *performance* of a design because it allows a designer to separate the performance critical parts from the non-critical ones. Then, the critical parts can be optimized in isolation, increasing their parallelism or using some techniques available for elastic systems, such as variable-latency units (e.g. using telescopic units [11], which can be naturally integrated into an elastic system). On the other hand, non-critical parts can have a relaxed timing and a long latency, reducing power consumption without a big impact on the overall performance. Furthermore, *power consumption* may also be reduced because the distributed localized control of elastic systems avoids unnecessary communications and computations.

Variability is one of the aspects with increasing importance in chip design [13, 117]. There exists variability among different chips of the same design (inter-die variability) due to imperfections in the manufacturing technology. As wires and transistors become smaller and smaller, the difference in the speed of the circuit due to these imperfections grows larger. Furthermore, there is also variability within a single chip (within-die variability). Different parts of the chip may be working at different operating conditions (e.g., at different temperatures), and hence, they have a different timing. Unpredictable delays compel designers to use conservative margins that cannot be adjusted in order to achieve the full potential of current process technologies.

Elasticity can tolerate timing variations in the computations and communications of a circuit and its environment, even if these variations are unpredictable. This tolerance may be used to adapt the voltage and speed of the circuit to the operating conditions without affecting the correctness of the computations. The intrinsic modularity of elastic systems also guarantees that changes in the speed of one module do not affect the correctness of the inter-module interaction.

The impact of wire delay on design performance has been growing on deep submicron design, due to the mismatch in scaling trends of gate delays and interconnect delays over process generations [3, 47]. The maximum distance a signal can travel along an interconnect within a clock cycle gradually decreases as the process technology scales down [138]. Traditional design methods require knowing communication delays in advance, but the distance between two locations, and hence the wire delay, cannot be accurately estimated on early design stages because the floor-plan is not available.

Elasticity can help to cope with *long communications* because it is possible to insert empty sequential elements, or *bubbles*, without affecting the correctness of the design. Hence, they can be added at any design stage to cut a long wire. While bubbles can solve timing problems due to long wires, they can also decrease the overall throughput of the system, since communications will have a larger latency. Hence, they must be added carefully to avoid a dramatic performance

penalty.

Current chips have the capacity to contain billions of transistors, but there is a gap between such manufacturing capabilities and the ability of designers to quickly put together a complex device. Reuse of existing intellectual property (IP) can help closing this gap and meeting tight time to marked constraints. As the number of IP elements on a single chip increases and the SoC market becomes more important, methods that allow faster design and re-usability of IP elements are needed.

Elastic systems can also help *reusing and integrating IP blocks*, since they define a simple communication protocol which can be used for synchronization and data transfer of already existing blocks. The functionality of elastic systems does not depend on the timing of its modules, only on the order of the transmitted data. Thus, it becomes easier to integrate modules with different timing. Since elastic controllers are distributed and there are no global signals to broadcast, this paradigm scales well to a large number of integrated IP blocks.

1.5 Overview of the Contributions

Elasticity opens the door to a new avenue of correct-by-construction behavior-preserving transformations for optimizing systems that cannot be systematically applied in a non-elastic context. These transformations take advantage of the tolerance of elastic systems to latency changes, and the fact that stalls and data synchronization is handled inherently by elastic controllers. The insertion of "empty" sequential elements in a pipeline, the execution of variable-latency computations, the addition of bypass circuits with early-evaluation firing semantics or the speculative execution of operations are some of the transformations that can be considered to increase the performance of a circuit.

The main objective of this thesis is to leverage these transformations in order to provide an optimization framework for elastic circuits. The contributions of this thesis are the following:

1. **A collection of correct-by-construction transformations applicable to elastic systems.** Some of these transformations can be applied to conventional synchronous systems, and they have been adapted to elastic systems. The rest of the transformations can only be applied in an elastic environment because they modify the latency of the system. This collection is the starting point of the optimization framework proposed in this thesis.

   It is verified that each of the transforms does not change the functionality of the design. The

timing of the events may change, but their order and their correct synchronization is always preserved.

All of these transformations can be applied to asynchronous circuits as well, but this thesis focuses on synchronous elastic systems, since verification and synthesis tools are better prepared to handle synchronous circuits and it is more practical to work with them.

2. **A method to introduce speculative execution in elastic systems.** This contribution shows how speculation can be inserted in an elastic system by applying a sequence of correct-by-construction transformations from the available collection of transforms. Speculation is a widely-used technique that increases the parallelism of a design by starting computations that are likely to be useful before it is known whether they will be needed.

   When speculating on a choice in an elastic system, one of the possible outcomes is predicted, based on some knowledge of the system, and the other possible choices are stalled. Such stalling is handled naturally by the handshake controllers of elastic systems. Then, normal computation continues with the predicted choice.

   Once the outcome of the prediction is known, the stalled data values can be discarded if the prediction was right, or used if the prediction failed. On mispredictions, the speculative execution is canceled. Assuming most of the times the predictions are successful, correction computations will be rarely needed, and the overall performance of the system will be better compared to a non-speculative execution.

   Several applications of this method are shown, such as branch prediction, error detection and correction units and an efficient implementation of variable-latency units.

3. **Automatic correct-by-construction pipelining.** One of the significant choices made during the early design stages is the structure of the pipeline. Building an efficient pipeline is a challenging problem [75]. However, it is often done ad hoc, due to the significant computational costs of simulation during the design space exploration and the lack of analytical optimization methods capable of pipelining in the presence of dependencies between iterations.

   An elastic system can be pipelined (even in presence of cycles and dependencies between iterations) by applying a sequence of correct-by-construction transformations. By using the appropriate kit of elastic transformations, different architectures can be derived for the same

functionality, thus enabling the capability of creating microarchitectural exploration engines that can guide the design of high-performance and low-power circuits.

Starting from a functional representation of the microarchitecture or a partially pipelined design, and a description of the expected workload on the design, this contribution presents an automatic method to perform design space exploration of elastic systems, in order to produce optimal or near-optimal pipelines for the expected workload. The resulting pipeline is optimized for the average case in terms of data variability.

The pipelining engine combines heuristic methods with a formal mathematical formulation that integrates most of the transformations presented in the first contribution. In order to avoid simulations with a heavy computational cost, the performance of the explored design points is approximated with analytical methods. Then, the most promising points are simulated in order to measure their performance more accurately. The outcome of this method can also be a set of Pareto points with different clock cycles and throughputs such that a designer or an architect can select the best suited for the target application. Manual crafting of an interlocked pipeline can be laborious and error prone, while the exploration engine presented in this thesis guarantees a provably-correct construction.

4. **A method to evaluate the performance of an elastic systems with early evaluation and variable latency.** The performance of an elastic system with early evaluation cannot be exactly analyzed using an efficient method. Current methods allow to compute upper and lower bounds using linear programming. However, the lower bound ignores early evaluation, and hence, provides poor results. The exact performance can be evaluated by techniques that suffer from exponential state explosion.

   This thesis presents a fast analytical method to obtain a lower bound of the performance of an elastic system with early evaluation and variable-latency units. This bound is much tighter than the previously known bound. This method is based on a symbolic analysis of the structure and the timing of the elastic system that is being evaluated. A better lower bound analysis method helps to improve the results of the automatic exploration engine since performance estimations become more accurate.

1.6 Organization of the Thesis

This document is organized as follows:

- **Chapter 2** reviews some basic concepts that will be used along the document, such as Petri net theory. Next, an introduction to latency-insensitive systems and synchronous elastic systems is provided. Finally, the previous work on automatic pipelining is reviewed.

- **Chapter 3** presents a set of correct-by-construction behavior-preserving transformations for optimizing elastic systems, which corresponds to the first contribution of this thesis. It is shown how each transformation can help to improve the overall performance of an elastic system.

- **Chapter 4** introduces speculative execution in elastic systems by using correct-by-construction transformations, which corresponds to the second contribution of this thesis. Several design examples are provided to illustrate the utility of speculation applied to elastic systems.

- **Chapter 5** discusses an automated microarchitectural exploration engine which allows to pipeline an elastic system automatically. This method corresponds to the third contribution of this thesis. Several pipelining examples are presented, and it is shown how pipelines with a near-optimal performance can be achieved given the expected workload of the design.

- **Chapter 6** proposes a method to analyze the performance of an elastic system using symbolic expressions, corresponding to the fourth contribution of this thesis. Two versions of the method are presented: an exact method that has high run-time complexity and an efficient approximate method that computes the lower bound of the system throughput.

- **Chapter 7** concludes the thesis, summarizing the main results and indicating possible directions for future research.

Chapter 2

State of the Art

This chapter starts with preliminary definitions of some basic concepts that will be used throughout this work. Then, it continues with an overview of synchronous elastic systems. All the contributions of this thesis rely on the capabilities provided by such systems, where the relationship between the system clock and data is loosened. Finally, a review of techniques for automatic pipelining is presented.

2.1 Preliminaries

2.1.1 Petri Nets

Petri nets are a well known mathematical model use to describe distributed systems. A Petri net (see [115,142] for tutorials) is a directed bipartite graph formed by *transitions*, which model events and are drawn as a bar, and *places*, which model conditions and are drawn as a circle. The arcs of the net connect transitions to places and places to transitions. Places can contain a natural number of *tokens*. A marking assigns a number of tokens to each place.

More formally, a Petri net is a 4-tuple $P = \{P, T, F, M_0\}$ where:

Figure 2.1: Petri net before and after firing transition t.

- P is a finite set of places, $p_1, p_2, \ldots, p_m$,
- T is a finite set of transitions, $t_1, t_2, \ldots, t_n$,
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs,
- $M_0 : P \longrightarrow \mathbb{N}$ is the initial marking, which assigns a number of tokens to each place.

The preset of a transition t is the set formed by its input places, ${}^{\bullet}t = \{p \in P \mid (p, t) \in F\}$, and its postset is the set formed by its output places, $t^{\bullet} = \{p \in P \mid (t, p) \in F\}$.

A transition t is said to be *enabled* if each of its input places is marked with at least one token. When an enabled transition *fires*, one token is removed from every input and one token is added to every output. Figure 2.1 shows the marking of a Petri net before and after firing transition t.

After a transitions fires, the marking M is transformed to a new marking M' ($M \rightsquigarrow M'$). A marking M' is reachable from another marking M if M' can be achieved after a sequence of transition firings starting on M, i.e., there exists markings $M_1, M_2, \ldots$ such as $M \rightsquigarrow M_1 \rightsquigarrow M_2 \ldots \rightsquigarrow M'$.

A Petri net is said to be *live* if from any reachable marking, for every transition it is possible to reach a marking where this transition can be fired. That is, if all transitions can continue being fired from any of the reachable markings. Liveness is related to the absence of deadlocks and livelocks.

2.1.2 Marked Graphs

One of the most interesting subclasses of Petri nets are *marked graphs* (MGs), where every place has exactly one input arc and one output arc ($\forall p \in P,\ \mid {}^{\bullet}p \mid = 1 \ \wedge \ \mid p^{\bullet} \mid = 1$). One important property of MGs is that they are live if and only if the sum of tokens on each simple cycle is greater than zero. In an MG there cannot be a conflict because there is no choice, each place can only provide tokens to one transition. On the other hand, MGs are a very good model for concurrent systems. For example, they can be used to analyze asynchronous designs and synchronous elastic systems.

For performance evaluation purposes, it is interesting to add time information to marked graphs. A timed marked graph (TMG [115]) is a marked graph plus a function δ which assigns a non-negative delay to every transition ($\delta : T \longrightarrow \mathbb{R}^+ \cup \{0\}$). Once a transition t of a TMG is enabled, it will fire after $\delta(t)$ time units.

Transitions can have infinite server semantics, meaning that they can fire multiple times on the same time unit (given that the input places have multiple tokens available); or they can have

single server semantics, where each new firing can only begin once the previous firing has been completed. Most evaluation techniques consider infinite server semantics [29]. Single server semantics can always be enforced by adding a self-loop with one token to each transition.

2.1.3 Throughput of a TMG

Given a transition t of a TMG, its throughput $\Theta(t)$ is defined as the average number of times t fires per time unit in an infinitely long execution of the system:

$$\Theta(t) = \lim_{\tau \to \infty} \frac{\sigma_t(\tau)}{\tau}$$

where τ represents time and $\sigma_t(\tau)$ the number of times that t has fired after τ time units. On an infinitely long run of a strongly connected TMG the throughput of all transitions is the same. The throughput of a synchronous elastic system can be found by evaluating the throughput of the TMG that models the elastic system.

2.1.4 Linear Programming

Linear programming [116] (LP) is a fundamental mathematical method used to optimize a linear objective function given a set of linear constraints. It has a large number of applications, including performance analysis of synchronous elastic systems. All linear programming problems can be expressed in a simple canonical form:

$$\begin{aligned} \text{maximize:} \quad & c^T \ x \\ \text{subject to:} \quad & Ax \leq b \end{aligned} \tag{2.1}$$

where x represents the vector of variables to determine, c and b are vectors with known coefficients, and A is a matrix of known coefficients. The expression to optimize ($c^T \ x$) is called the objective function. The assignment on the vector of variables x that maximizes the objective function is the *optimal value*. If an assignment satisfies all the constraints it is called a *feasible solution*.

LP problems can be solved efficiently using several available methods. The most popular is the simplex algorithm [56]. Linear programming can be solved in polynomial time [88, 90], but it becomes NP-complete is a subset of the variables is forced to have an integer value [125]. This last problem is called integer linear programming (ILP).

2.2 Synchronous Elastic Systems

Latency-Insensitive Designs [31, 32] emerged as a methodology to design complex systems by integrating a set of IP blocks. These IP blocks operate like a regular synchronous module, but they can be stalled if some inputs are not available or when data receivers are not ready.

Latency-insensitive designs are also nominated as synchronous elastic designs [53] pseudoasynchronous design [94], synchronous handshake circuits [127], interlocked pipelines [82] or synchronous emulation of asynchronous circuits [123].

In [49, 53, 96], Synchronous Elastic Systems are presented as a possible efficient implementation of latency-insensitive designs. The Synchronous ELastic Flow (SELF) [53] protocol is similar to the latency-insensitive protocols presented in [82, 127] and it is conceptually similar, but not equivalent, to the one presented in [31]. SELF designs typically target more fine-grain latency-insensitive designs, while other protocols target elasticity at a coarser granularity, for example for synchronization of IP blocks. This thesis focuses on synchronous elastic systems to develop all its contributions, although it should be possible to apply them to any latency-insensitive protocol and to some classes of asynchronous systems.

The following sections provide an overview of synchronous elastic designs. First, the basic definitions of the methodology used in latency-insensitive and in synchronous elastic designs are presented. Then, the SELF protocol and a possible implementation of its controllers are presented. Next, early evaluation and token counterflow are introduced, and verification of elastic controllers is discussed. After that, it is explained how elastic systems with no early evaluation can be scheduled in order to reduce routability problems. Thereafter, the possible granularities of elasticization and a synthesis flow for synchronous elastic systems are presented. Finally, it is explained how marked graphs can be used to model elastic systems and how the performance of elastic systems can be analyzed.

2.2.1 Definitions

Elastic Module

The elastic modules of an elastic system operate like a regular synchronous block, but they can be stalled if not all inputs are available or when data receivers are not ready. They can be any building block of a regular synchronous design, such as an ALU or any other IP block.

In order to be assembled into a latency-insensitive design, modules must be *stallable*, which

means that they must be able to be stalled for an arbitrary number of clock cycles without losing their internal state. Stalling can be easily achieved with latched blocks and standard clock gating techniques.

Using latency-insensitive terminology, modules are encapsulated within a wrapper (called shell) that communicates with the channels. The shells that encapsulate the modules of a latency-insensitive design are *patient*, meaning that the system behavior only depends on the order of the events of the signals and not on their exact latency.

Elastic Channel

A channel is comprised from a set of data wires and a few control signals implementing a communication handshake. The handshake signals communicate using a certain protocol in order to decide whether the data sent through the channel is meaningful and can be transmitted. This handshake is synchronous in the case of latency-insensitive designs or asynchronous in the case of asynchronous designs.

When a channel transmits some piece of data, it is said that the channel transmits a *token*. If no tokens are sent through the channel, then it sends a *bubble*. Data communication may not be possible because the sender has no data to transmit or because the receiver is not ready. In the second case, it is said that the receiver introduces *backpressure* in the channel in order to stop the sender.

Elastic Buffer

Elastic buffers (EB) store and transmit tokens through elastic networks. EBs are called *relay stations* in the terminology of latency-insensitive designs [31]. An elastic buffer is conceptually similar to a flip-flop, with some logic gates in the control to implement the elastic communication protocol.

More generally, an EB can be defined as a FIFO, which has the capacity to store a limited number of tokens (or data items). Furthermore, its latency is the number of clock cycles that a token needs to travel from its input to its output when there is no backpressure. If an EB contains no tokens initially, then it is called a bubble[1]. Bubbles are patient processes that initially contain no data, and hence, they can be inserted within any channel and the resulting system will be

[1]Notice that bubble has several meanings. It is said that a channel transmits a bubble when it has no data to transmit. A buffer with no data is also called a bubble

Synchronous	a_1	a_2	a_3	a_4	$\ldots$	a_k	$\ldots$				
behavior:	b_1	b_2	b_3	b_4	$\ldots$	b_k	$\ldots$				
Elastic	a_1	$*$	a_2	$*$	$*$	a_3	$*$	a_4	$\ldots$	a_k	$\ldots$
behavior:	b_1	$*$	$*$	b_2	$*$	b_3	b_4	$\ldots$	b_k	$\ldots$	

Figure 2.2: Behavior of a design with two internal registers, *a* and *b*. At each cycle, a different value is stored at the registers. In the elastic behavior, the symbol '*' denotes cycles with non-valid data (bubbles).

functionally equivalent. For example, they can be used to cut long interconnection delays in any stage of the design flow [31]. In section 2.2.2, EBs are defined formally within the SELF protocol.

Transfer Equivalence

Two sequential designs are considered behaviorally equivalent if they produce the same output stream when they receive identical input streams. In a conventional synchronous design, this equivalence holds cycle-by-cycle, i.e. two designs are equivalent if after receiving the same stream $x_0, x_1 \ldots x_k$, they produce an identical output stream $z_0, z_1 \ldots z_k$, where x_i and z_i represent the input and output values at cycle i respectively.

In elastic systems, cycle count is decoupled from data count, and comparing cycle-by-cycle streams does not make sense. Instead, sparse streams of data with void cycles (bubbles) are produced. For elastic designs, more general notions of equivalence have been defined in a few slightly different frameworks: latency equivalence [32], flow equivalence [69], or transfer equivalence [96].

These equivalences guarantee that for every output of the design, the order of valid data items is the same as in the conventional synchronous design when equivalent input streams are applied, i.e., the elastic and non-elastic behaviors are indistinguishable after hiding the bubbles in the elastic streams. For example, the synchronous behavior and the elastic behavior in Fig. 2.2 are transfer equivalent. This notion of equivalence enables a larger spectrum of transformations for design optimization.

Throughput

Data in an elastic system may not be transferred during all clock cycles. The *throughput* of an elastic channel is the average number of tokens processed during a cycle (a metric similar to IPC,

instructions per cycle, in a CPU). Given a stream of a channel, like the ones shown in Fig. 2.2, the throughput is defined as the percentage of valid tokens over the total length of the stream.

The throughput of an elastic system can be obtained by performing a simulation which is long enough, and counting how many tokens are transferred in any of the channels of the design[2]. Alternatively, there are analytical methods that can compute the throughput of an elastic design. Section 2.2.9 provides more details on how the throughput can be evaluated using analytical methods.

Effective Cycle Time

In synchronous designs, the cycle time is an important metric since it defines the maximum clock frequency. The cycle time is the delay of the longest combinational path in the design. In an elastic design, the *effective cycle time*, the cycle time divided by the throughput, is the average time elapsed between two token transfers in a channel and is similar to an average time per instruction in processors. The effective cycle time is the main optimization target on elastic systems.

2.2.2 SELF Protocol

SELF defines a formal protocol for creating an elastic system. A pair of control signals bits (*valid* and *stop*) implements a handshake protocol between the sender and the receiver of an elastic channel. The valid bit, going in the forward direction, is set by the sender when some piece of data (a *token*) is being sent. The stop bit, going in the backward direction, implements *backpressure* and is used for stalling the sender when the receiver is not ready.

The protocol allows three possible states in an elastic channel, as shown in Fig. 2.3:

(T) Transfer, ($V \wedge \overline{S}$): the sender provides valid data and the receiver accepts it.

(I) Idle, ($\overline{V}$): the sender does not provide valid data.

(R) Retry, ($V \wedge S$), the sender provides valid data but the receiver does not accept it.

The language protocol observed on the pair (V, S) is described by the regular expression (I*R*T)*. In can be proven that a particular implementation complies with this protocol by verifying the following properties, expressed in *linear temporal logic* [128] (LTL):

[2]The throughput of all the channels should be the same after enough cycles

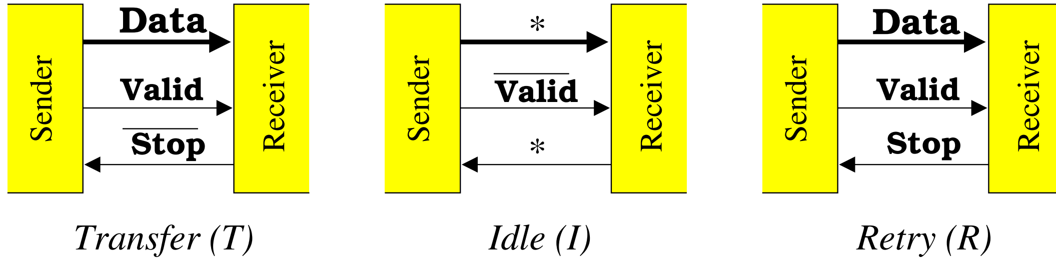


Figure 2.3: The SELF protocol.

$$\mathsf{G}\,((V \wedge S) \implies \mathsf{X}\,V) \qquad \text{(Retry)}$$

$$\mathsf{G}\,\mathsf{F}\,(V \wedge \overline{S}) \qquad \text{(Liveness)}$$

The first property states that the sender of a token has a *persistent* behavior. Whenever a *Retry* cycle is produced, valid data items are held until a transfer occurs. The second property states that the channel must be live, a token must always be eventually sent. These two properties are true if and only if the regular expression (I*R*T)* is true.

Elastic Buffers in SELF

In general, an EB is a FIFO with an arbitrary finite capacity. An abstract model for an EB is modeled in Fig. 2.4. An EB is an unbounded FIFO that complies the SELF protocol at both input and output channels. B is an infinite array that stores data into the buffer. The write pointer (wr) points to the position where the next data item will be written, and the read pointer (rd) points to the position where the token that is being transmitted is stored. The value $\mathsf{k} = \mathsf{wr} - \mathsf{rd}$ is the current number of tokens in the buffer.

The notation X_{nxt} is used to represent the next-state value of variable X. The symbol $*$ represents a non-deterministic value (don't care).

The retry variable stores whether the output channel attempted a transfer in the previous cycle. If retry is true, the same data item as in the previous cycle is issued and V_{out} must be set in order to properly follow the SELF protocol. If the buffer does not contain any tokens, no transfer can be performed ($V_{out} = false$). Otherwise, V_{out} can be any value, depending on the delay of the FIFO. S_{in} can non-deterministically stop any data transfer at the input channel. The items stored in the

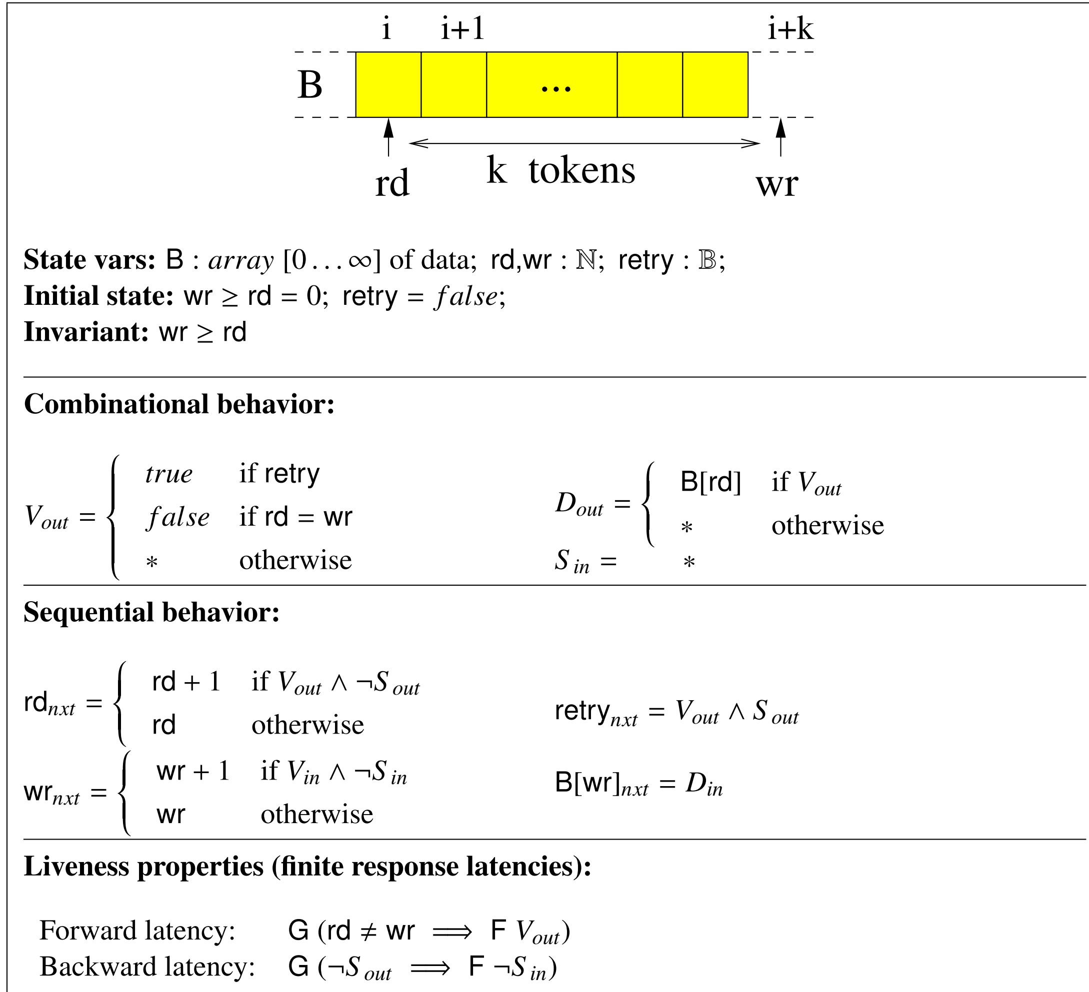


Figure 2.4: Abstract model for an elastic FIFO.

buffer are eventually transferred to the output after a finite unknown delay. Two liveness properties expressed in linear temporal logic ensure finite response time: (1) data from the EB will eventually be sent to the output, and (2) a non-stop at the output will eventually be propagated to the input.

The capacity C of an EB defines the maximum number of tokens that can be stored inside the buffer simultaneously. The model in Fig. 2.4 can be bounded with capacity C by changing the behavior of S_{in}: $S_{in} = (\mathsf{wr} - \mathsf{rd} = C)$. It is known that the following constraint must be satisfied: $C \geq L_f + L_b$, where L_f is the forward-propagation latency (the number of clock cycles required to propagate data and the valid bit from the input to the output) and L_b is the backward-propagation latency (the number of clock cycles required to propagate backpressure from the output to the input). This constraint has been proved in [32, 102]. Typically, $L_f = L_b = 1$ and $C = 2$, which

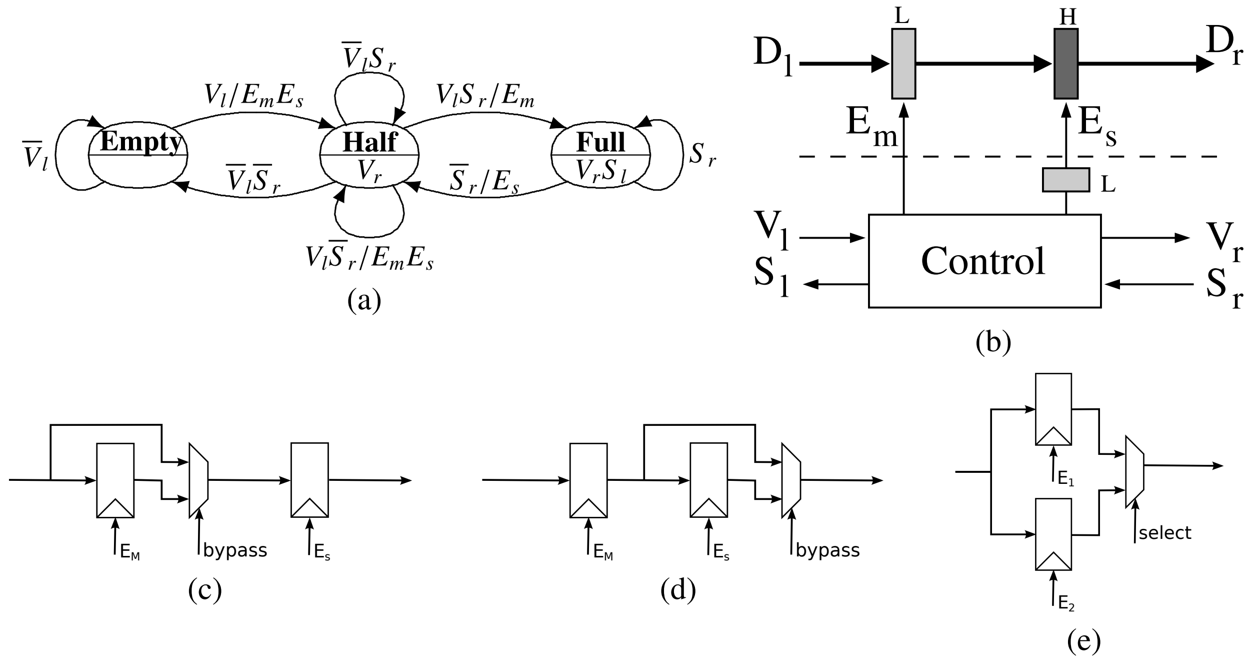


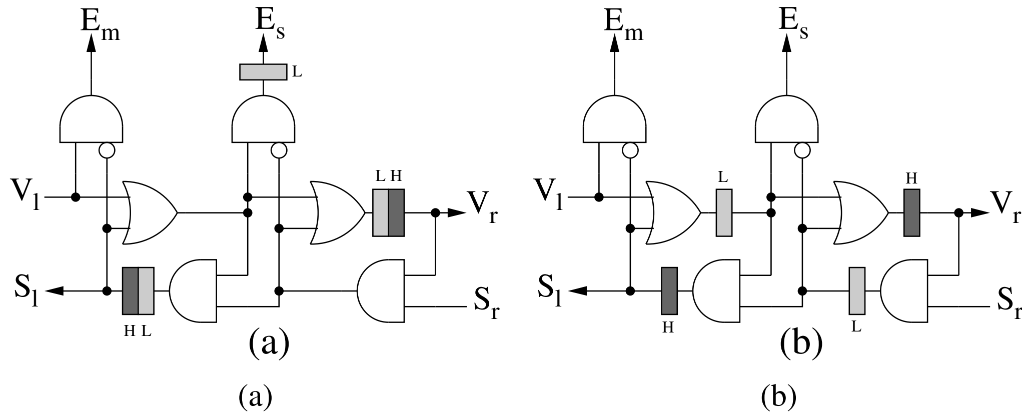
Figure 2.5: (a) FSM of an elastic buffer, (b) how control is connected to the datapath, (c),(d),(e) alternative datapath implementations for the elastic buffer in the FSM.

corresponds to the case where the EB behaves exactly as a flip-flop when there is no backpressure and a constant flow of tokens arrives at the input.

Multiple constructions of elastic FIFOs have been studied in asynchronous and synchronous design communities (see, e.g., [43]). Each possible implementation provides a different area, power, delay trade-off. An efficient latch-based implementation for the case in which $L_f = L_b = 1$ and $C = 2$ is presented in [53] for SELF. Figure 2.5(a) shows the FSM specification for the control of this latch-based EB and Fig. 2.5(b) shows how this control is connected to the datapath. The datapath is implemented using transparent latches, which have been drawn as boxes labeled with their phase (active low L or active high H). An elastic buffer can be viewed as a composition of two elastic *half buffers*, the active low one and the active high one. The enable signals produced by the control are used to clock gate the datapath latches.

The FSM specification shows three states that correspond to the cases in which the EB contains no tokens (*Empty*), contains one token (*Half*), and contains two tokens (*Full*). Notice that the initial state can be any of them, depending on the number of tokens initially in the EB.

Figures 2.5(c), 2.5(d) and 2.5(e) show other possible ways to implement the EB specified in the FSM using only flip-flops. The control, which is not shown in these figures, would be responsible



(a) (b)

Figure 2.6: Control implementations of an elastic buffer (a) using either flip-flops or latches, (b) using only latches.

for generating the enable signals of the flip-flops and the select signal of the multiplexor. The implementations in Fig. 2.5(c) and Fig. 2.5(d) use two flip-flops connected in a sequence. When there is no backpressure, the multiplexor must select the bypass path to allow a regular flow of tokens. If the FSM goes into the *Full* state, the extra flip-flop must be enabled to store the second token. When emptying the EB, the select signal of the multiplexor must make sure that the order of the tokens is not changed. In Fig. 2.5(c) the extra flip-flop is the first one of the sequence, while in Fig. 2.5(d) it is the second one. Figure 2.5(e) uses two flip-flops in parallel. One of them is used regularly, and the second one is only used when two tokens must be stored. In general, the implementation in Fig. 2.5(b) is better in terms of area and power, as long as there are latches available in the implementation library.

Figure 2.6 shows two possible implementations of the controller for the datapath in Fig. 2.5(b). Figure 2.6(b) uses only latches, but Fig. 2.6(a) can be synthesized using either latches or flip-flops, as the pair of latches are connected with no gates in between. The EB controller is formed by two elastic half buffer controllers, identical but with inverted latch polarities. In order to show correctness of the implementation, it can be proven that either one of the controllers shown in Fig. 2.6 is a refinement [109] of the FSM in Fig. 2.5(a), and that the FSM is a refinement of the abstract model presented in Fig. 2.4. Similar controllers can be generated for the other implementations shown in Fig. 2.5.

In some special cases, it is useful to use 0-latency buffers ($L_f = 0$), or *skid-buffers*. A skid-buffer behaves like a regular elastic channel while the receiver is ready: all input tokens are propagated combinationally. However, the skid-buffer behaves like an elastic buffer when the receiver

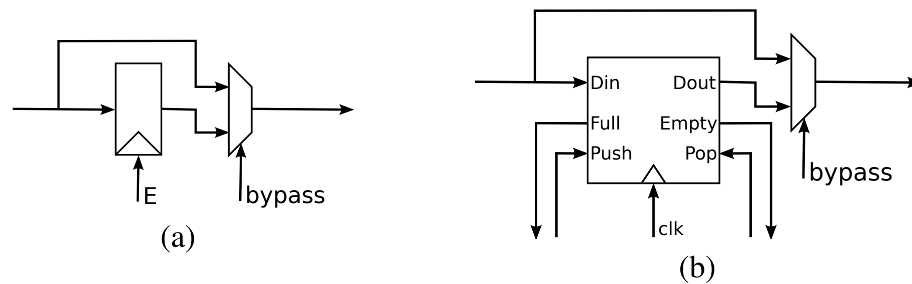


Figure 2.7: (a) Simple skid-buffer of capacity 1 using a flip-flop, (b) skid-buffer using a synchronous FIFO.

applies backpressure to the channel,

Figure 2.7(a) shows the datapath implementation of a simple skid-buffer with capacity to store one token. When there is no backpressure, the multiplexor just selects the bypass path, and the input token is propagated to the output. On the other hand, when the stop bit is asserted, the control must enable the flip-flop in the skid-buffer, and the multiplexor must change its selection in order to preserve the order of the tokens. This skid-buffer implementation can be extended to larger capacities. However, if the capacity of the skid-buffer is big enough, it may be more efficient to use a regular synchronous FIFO with latency 1, as shown in 2.7(b). In this case, the elastic control must drive the bypass multiplexor and the control signals of the FIFO (*Push* and *Pop*). The *Empty* and *Full* signals must be used to compute the valid and stop signals of the controller. If the FIFO is full, the input stop signal must be asserted, and if the FIFO is empty and no token arrives from the input, the output valid signal must be 0.

Forks and Joins

Elastic buffers can be generalized to any number of input and output channels by connecting additional control in the inputs and the outputs. This control must be able to generate one single pair of valid and stop bits from several pairs of them, and to generate several valid and stop bits from a single pair of them.

Figure 2.8(a) shows the control for a join structure. A pair of channels is merged into a single channel. The control transmits a token forward when both channels have a token, i.e., it waits for both senders to be ready. The input channels are stopped if they have a token to transmit (their valid signal is asserted) but the token cannot be transmitted through the output channel, either because some of the other inputs is not valid or because the stop bit of the output channel is true.

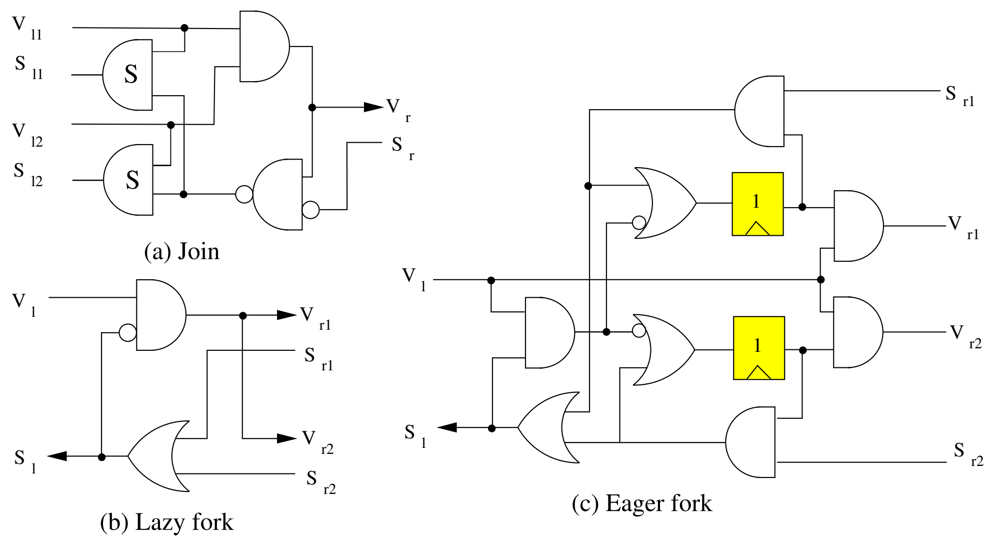


Figure 2.8: Control implementations of (a) an elastic join, and (b,c) an elastic fork.

It is possible to achieve a simpler implementation of the join by stopping the input channels whenever the transfer at the output channel is not possible. This way, the AND gates labeled with an *S* in Fig. 2.8(a) can be removed. However, this implementation allows to stop channels that are not valid, i.e., it is possible that the handshake signals of a channel are $\overline{\text{Valid}} \wedge \text{Stop}$. While this is not forbidden by the protocol shown in Fig. 2.3, it is cleaner to stop channels only when they are actually trying to send data.

Figures 2.8(b) and (c) show the control for two possible implementations of a fork. The lazy fork waits for both receivers before sending a token. If a receiver is ready and the other one is not, the ready receiver has to wait until the other one is prepared. On the other hand, the eager fork allows to send a token to the first receiver that is prepared while waiting for the other. Once the token has been sent to both receivers, the system can go on to the next token.

Notice that a join connected to a lazy fork with no elastic buffer in between creates a combinational loop. The eager fork can be used instead to avoid combinational loops. The eager fork implementation shown in Fig. 2.8(c) has one flip-flop for each output channel. This flip-flop is set false if and only if the corresponding output channel has transmitted a token but this token has been stopped at some other output channel. Then, this channel must send bubbles until the other output channels have transmitted the token and the next evaluation can start. For some special cases, eager fork provides a higher throughput than lazy fork.

It is possible to create elastic modules with an arbitrary number of input and output channels

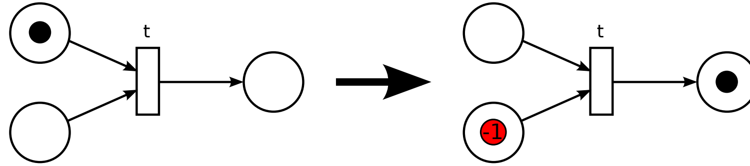


Figure 2.9: Early evaluation firing in transition t, an anti-token is added to the lower input place after firing.

by chaining several join or fork structures. There are available algorithms that allow to minimize the number of 2-input join and 2-output fork control components in order to optimize the control network [91].

2.2.3 Early Evaluation and Token Counterflow

[32, 34, 53] rely on lazy evaluation: the computation is initiated only when all input data are available. This requirement is often too strict. Consider a multiplexor with the following behavior:

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

If the value s is known, then it is only necessary to wait for the required token in order to compute o. If, for instance, s and a are available and the value of s is *true*, the result $o = a$ can be produced without waiting for b to arrive and the value of b can be discarded when it arrives at the multiplexor.

Early evaluation takes advantage of this flexibility to improve system performance. The irrelevant tokens with late arrival must be nullified in order to avoid spurious enabling of functional units. This is a key problem that must be solved correctly to guarantee that relevant data are not discarded by mistake. When early evaluation occurs, a negative token, also called *anti-token*, is generated in the late channels that were not used for enabling the block. When an anti-token and a token meet in the same channel, they cancel each other. Figure 2.9 shows early-evaluation firing semantics using a marked graph model. When transition t fires, it removes one token from each input and adds one token to each output. Since the lower input place has no tokens initially, an anti-token is added to ensure correct token ordering.

Anti-tokens can be *passive*, waiting for the token to arrive inside the early evaluation join controller, or *active*, traveling backwards through the control until they meet a token. Anti-tokens for early evaluation were already used in [92, 159], where Petri nets were extended to handle OR causality and nodes with arbitrary early-evaluation functions.

Several groups in the asynchronous design community have used early evaluation to allow a logic block to compute its outputs before all of its inputs are available. In [134], the inputs of logic blocks with early-evaluation are partitioned between early and late. Once all early inputs are available a new output may be produced. However, the block must still wait for all inputs to arrive before advancing to the next evaluation. This technique has also been applied to a coarser granularity [133]. In [6] and [21], anti-tokens flow in the backward direction and they cancel unneeded regular tokens. This token counterflow allows blocks to advance to the next evaluation without waiting all the inputs, achieving a higher performance. In [6], special care was taken to avoid metastability regardless of the arrival order of signals.

Several strategies have been proposed to incorporate early evaluation into latency-insensitive protocols [40, 49, 99]. A comparison between them can be found in [41]. A related method is proposed in [4,143] to optimize latency-insensitive systems in the presence of multi-clock domains.

In [99], a method to automatically derive don't care conditions of the datapath logic is proposed. For example, when an FSM is in a certain state, its state-transition and output functions may only depend on a subset of its inputs. Hence, it is not necessary to wait for the rest of the inputs. This functional independence conditions (or FICs) are then used to create early-evaluation logic that can boost-up the performance by avoiding unnecessary stalls.

In [40], latency-insensitive systems with early evaluation are called adaptive, as opposed to static systems when they are not early evaluated. In this work, an oracle obtains basic information taken from the logic block in order to select the necessary inputs for every new evaluation.

It can be considered that both [99] and [40] use passive anti-token implementations. Thus, early evaluation is handled locally, there is no extra communication between modules due to early enablings. When early evaluation is triggered in a module, its controller uses some internal counters to remember which channels are not valid. When a token arrives to an input channel with a counter that contains some anti-token, the token is discarded and the counter decreases its value. The maximum value allowed to this counters can affect the overall performance, because early evaluation cannot be allowed if it will cause an overflow on the counters.

In [49], SELF is extended to handle early evaluation both with active and passive anti-tokens. The conditions for early evaluation are not derived automatically, they must be provided by the designer. Using this method, the usual flow of tokens is complemented by a dual flow of anti-tokens going in the backward direction. Elastic buffers are extended so that they can store and propagate anti-tokens the same way they store and propagate tokens. Instead of two handshake signals, four handshake signals are required, two of them to propagate tokens in the forward direction and the

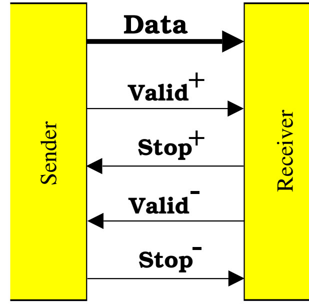


Figure 2.10: Elastic channel with dual protocol. Valid^{+} and Stop^{+} propagate tokens forward, while Valid^{-} and Stop^{-} propagate anti-tokens backwards.

other two to propagate anti-tokens in the backward direction, as shown in Fig. 2.10. The symbols V^{+}, S^{+}, V^{-} and S^{-} denote the *valid* and *stop* signals for the positive and negative flows. Note that the V^{-} signal has the semantics of a *Kill* for the positive tokens.

Protocol

The SELF protocol must be duplicated in order to be able to adopt token counterflow. The V^{+} and S^{+} signals implement the SELF protocol to transmit tokens and then V^{-} and S^{-} implement it to transmit anti-tokens in the opposite directions. In the boundaries of the EBs, tokens and anti-tokens must cancel each other when they meet.

Furthermore, a new invariant must be complied by the elastic channels: it is not possible to kill a token and to stop it at the same time. The same invariant applies to the dual channel:

$$\overline{V^{-} \wedge S^{+}} \quad \text{and} \quad \overline{V^{+} \wedge S^{-}} \qquad \text{(Dual Invariant)}$$

Persistency and liveness properties must also be extended for dual channels. A dual channel must comply the persistence property for both the forward pair of handshake signals and the backward pair of handshake signals, and it is live if there is always more tokens or anti-tokens to propagate:

$$\mathsf{G}\,((V^{+} \wedge S^{+}) \implies \mathsf{X}\,V^{+}) \qquad \text{(Retry+)}$$
$$\mathsf{G}\,((V^{-} \wedge S^{-}) \implies \mathsf{X}\,V^{-}) \qquad \text{(Retry-)}$$
$$\mathsf{G}\,\mathsf{F}\,((V^{+} \wedge \overline{S^{+}}) \vee (V^{-} \wedge \overline{S^{-}})) \qquad \text{(Dual Liveness)}$$

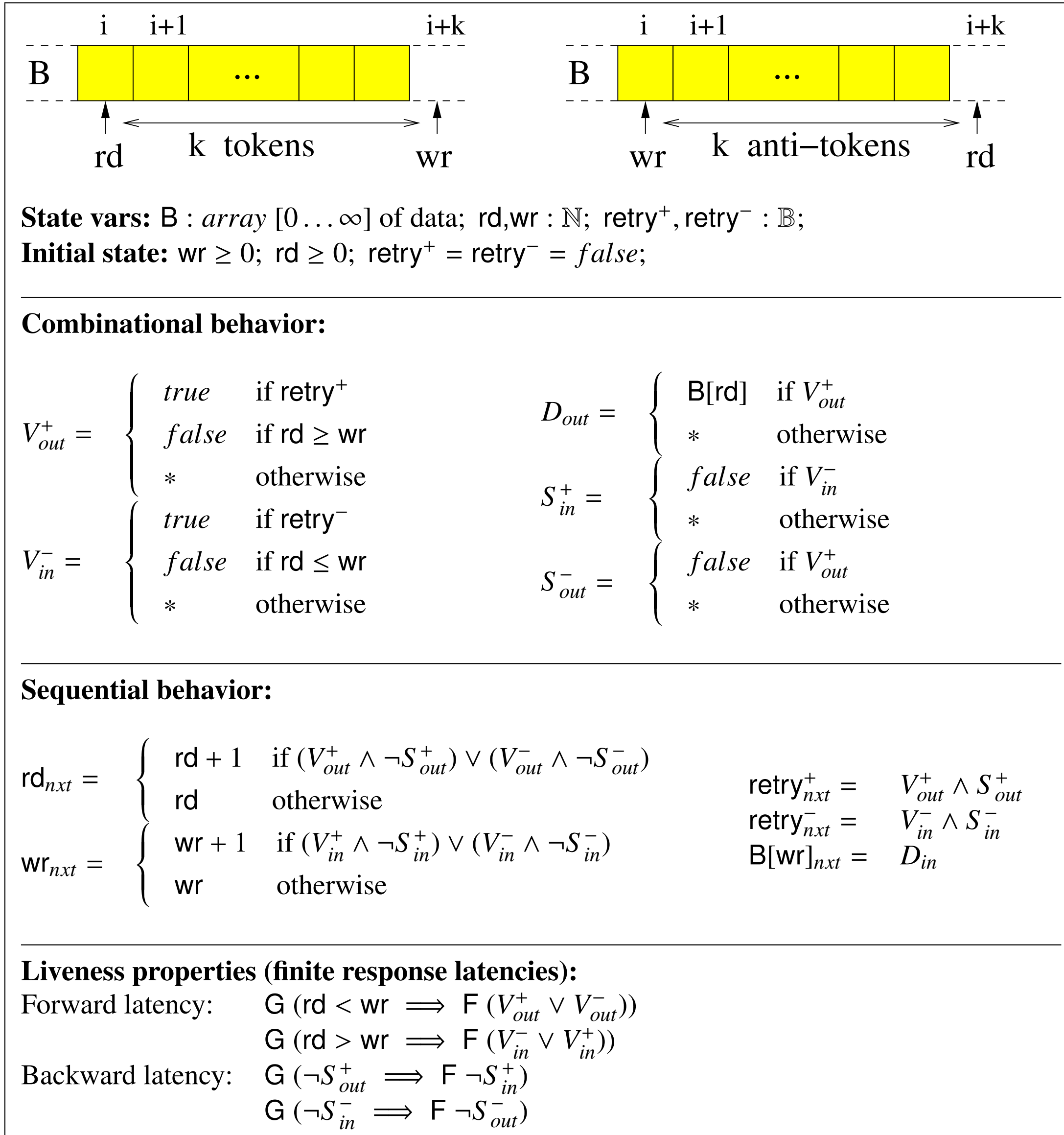


Figure 2.11: Abstract model for an elastic FIFO with anti-tokens.

Dual Elastic Buffers

The abstract model for an EB with token counterflow as described in Fig. 2.4 must be extended so that the buffer can store and propagate anti-tokens, as shown in Fig. 2.11. An EB becomes an unbounded FIFO which stores tokens (data items) and anti-tokens, which cancel each other at the boundaries of the EB.

B is an infinite array that stores the tokens written to the buffer. Variables wr and rd are the

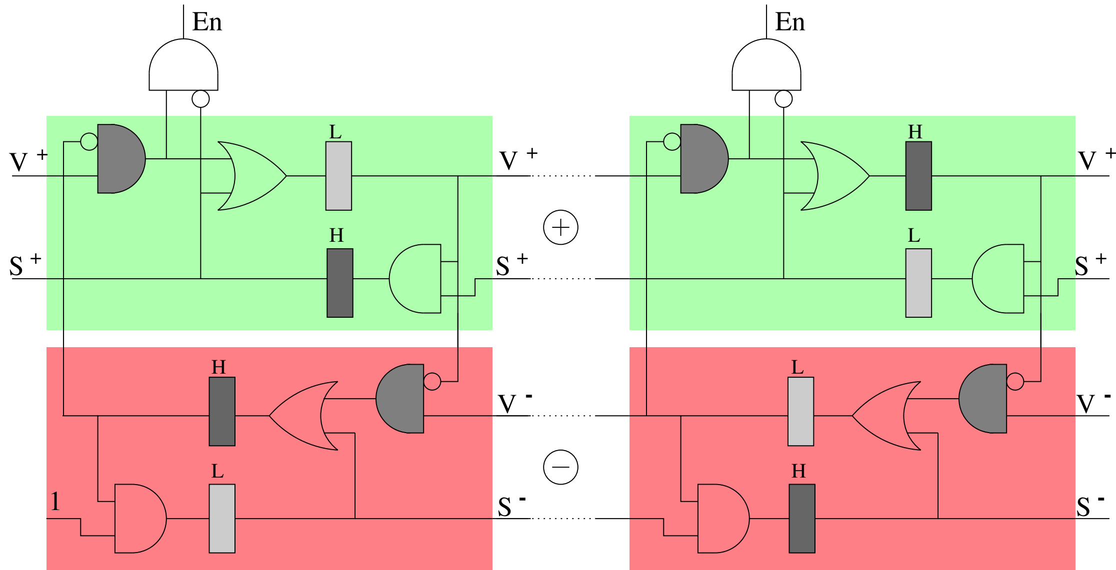


Figure 2.12: Linear pipeline with dual elastic buffers. The anti-token flow is stopped at the beginning of the pipeline by setting S^-_{in} to 1.

write and read pointers respectively. When $\mathsf{wr} > \mathsf{rd}$, the buffer contains $k = \mathsf{wr} - \mathsf{rd}$ tokens of data. On the other hand, when $\mathsf{wr} < \mathsf{rd}$, the buffer contains $k = \mathsf{rd} - \mathsf{wr}$ anti-tokens. Notice that when the buffer stores tokens, each incoming anti-token decrements the amount of tokens, while each incoming token increments the number of tokens. When the buffer stores anti-tokens it works in the opposite way. If $\mathsf{wr} = \mathsf{rd}$ the buffer is empty.

The retry^+ and retry^- variables ensure that transmissions are persistent, as required by the protocol. If the buffer contains no tokens (anti-tokens), no transfer can be performed, $V^+_{out} = false$ ($V^-_{in} = false$). The cases $V^+_{out} = *$ and $V^-_{in} = *$ model the non-deterministic latency of the buffer for propagating tokens and anti-tokens. The stop bits are asserted non-deterministically satisfying the constraint that both tokens and anti-tokens cannot be canceled and stopped at the same time.

The liveness properties expressed in LTL ensure finite but unbounded latencies in the forward and backward directions for both tokens and anti-tokens. The first property states that each token stored in the buffer must eventually be sent or killed by an anti-token (finite forward response). The second property ensures a finite response time of the stop bits: each time the receiver environment is ready to receive a new token, the buffer must eventually become ready to receive a new token from the sender. The two symmetric properties must hold for anti-tokens.

The implementation of regular elastic buffers, shown in Fig. 2.6, is substituted by the controller

in Fig. 2.12. In Fig. 2.12, the upper layer transmits tokens forward and generates the enable signals for the datapath latches, while the lower layer transmits anti-tokens backwards. The shaded AND gates cancel tokens and anti-tokens when they meet in the interface of an elastic half buffer.

Notice that token counterflow requires to duplicate the controller logic and hence the controller area. Therefore, active anti-tokens should only be used when they can actually provide a performance benefit compared to passive anti-tokens. In order to stop the token counterflow, the anti-token stop bit S^- is set to one and the valid bit V^- is not propagated backwards anymore, as shown in the left-most anti-token controller in 2.12. If needed, an anti-token counter similar to the ones proposed in [40, 99] can be introduced in any channel to increase the capacity of a buffer to store anti-tokens.

Forks, Joins and Early Joins

When propagating anti-tokens, the join and fork structures from Fig. 2.8 are substituted by the join and fork structures in Fig. 2.13. The join controller in Fig. 2.13(a) is composed by a join controller to transmit tokens in the forward direction and a fork controller to transmit anti-tokens in the backward direction. Similarly, the fork controller in Fig. 2.13(b) is composed by a fork controller to transmit tokens and a join controller to transmit anti-tokens backwards. The gates labeled with I in Fig. 2.13(a) ensure that the stop bit is not asserted at the same time that the valid bit of the opposite polarity, in order to comply with the Dual Invariant rule.

The fork structures used in these dual controllers are different from the ones shown in Fig. 2.8, although they are equivalent. The main difference is the semantics of the flip-flop at each output channel of the fork. While the flip-flop of the original implementation is 0 if and only if the token was transmitted in this channel but not in some other channel, in the dual implementation the flip-flop is 1 when the token could not be transmitted to the output channel, and hence the protocol is in retry state for this channel. This change simplifies the communication between token and anti-token transfers. The gate labeled B in Fig. 2.13(a) ensures that V^+_{out} is not true if the fork sending anti-tokens has some pending transfer. This way, the controller does not propagate tokens that must actually be canceled.

Besides fork and join structures, an special join controller, shown in Fig. 2.13(c), is needed in order to enable early evaluation. Consider, for example, a join controller for a 2-input multiplexor, which must be early evaluated. It would have three input channels: s, a, and b. The first channel would be associated with the select signal. Since every channel is elastic, it would also carry the V

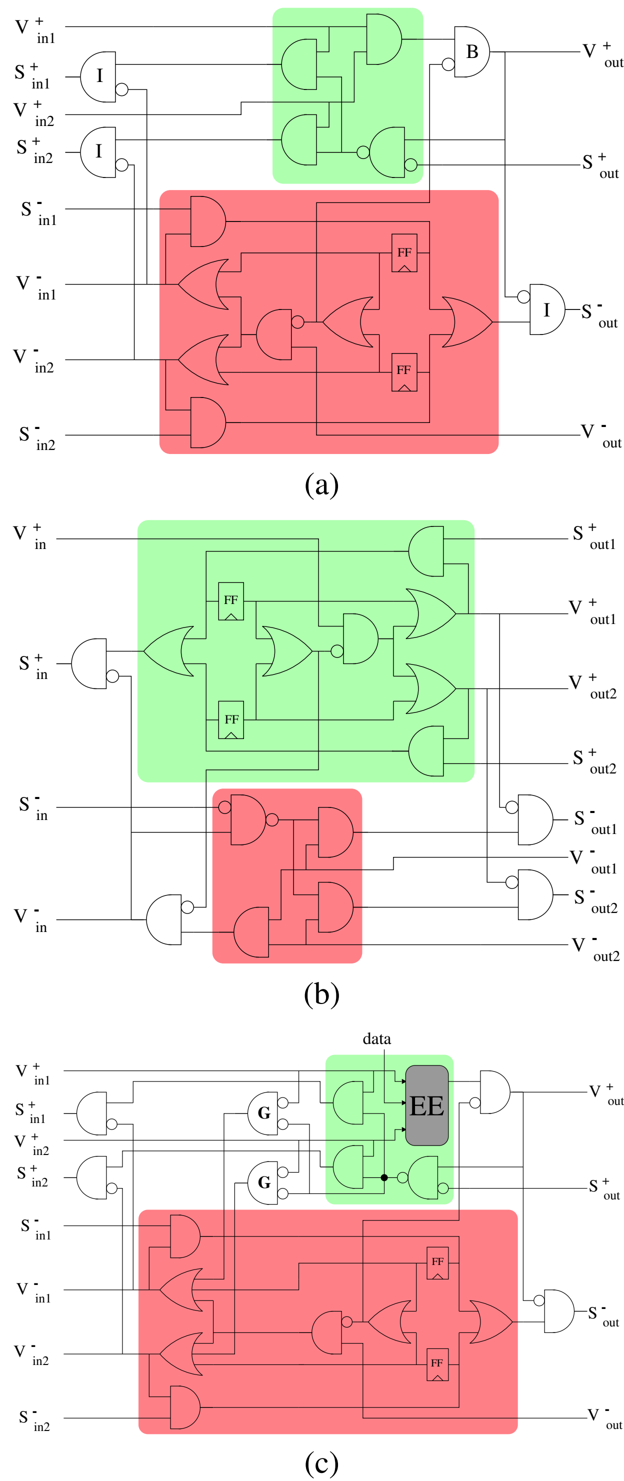


Figure 2.13: (a) dual elastic join, (b) dual elastic fork, (c) dual elastic join with early evaluation.

and S signals of the elastic protocol. The enabling function (block EE in Fig. 2.13(c)) would be:

$$\mathrm{EE} = V_s^+ \wedge ((s \wedge V_a^+) \vee (\overline{s} \wedge V_b^+)) \tag{2.2}$$

The signal s corresponds to the data value of the channel with the same name. Note that V_s^+ must always be true for the enabling of the module. An early enabling is produced, for example when $V_s^+ = V_a^+ = 1$ and $s = 1$. In this case, if $V_b^+ = 0$, an anti-token will be produced in the channel b. The gates labeled with G in the controller are responsible for generating such anti-tokens if necessary. These anti-tokens can travel backwards through a dual controller, or they can be stored locally in a counter. This controller can be optimized in some special cases by using token cages [38].

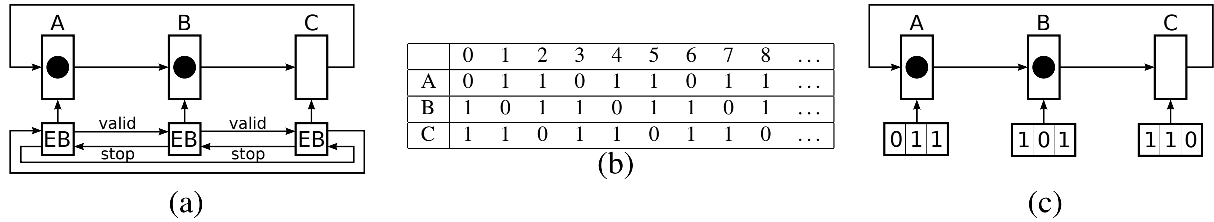
In order to ensure that the SELF protocol is complied by the controller, the cofactors of the EE function with respect to data inputs must be *positive unate* for the valid signals. In other words, EE can check for the presence of tokens, but never for their absence. For the previous example, the cofactors with respect to the data input s are:

$$\mathrm{EE}_{\overline{s}} = V_s^+ \wedge V_a^+ \qquad\qquad \mathrm{EE}_s = V_s^+ \wedge V_b^+ \tag{2.3}$$

Feedback paths from datapath to controller and back to the datapath must be designed carefully. If the output channel of an early-evaluation controller is connected to the input channel of an elastic buffer, a combinational path appears from the data input of the early-evaluation controller to the enable signal of the latch of the elastic buffer. Since the datapath latch is active low and the input probably comes from an active high latch or from a flip-flop, this combinational path must arrive in half cycle. Depending on the width of the EB (the number of actual one bit latches driven by this enable signal), the number of logic levels before the early join controller and the complexity of the EE function, this path might become critical. This potential problem must be considered when implementing early evaluation. The path from the valid bit to the enable signal of this critical latch can be removed to alleviate this problem. Then, the latch will always capture new data unless it is stopped, but correctness is still preserved.

2.2.4 Verification

Designing controllers for elastic systems is difficult and error-prone. It must be ensured that the designed controllers comply with the specified protocol (e.g., see [50,100]). This is typically done



	0	1	2	3	4	5	6	7	8	...
A	0	1	1	0	1	1	0	1	1	...
B	1	0	1	1	0	1	1	0	1	...
C	1	1	0	1	1	0	1	1	0	...

Figure 2.14: (a) Simple synchronous elastic pipeline, (b) Simulation trace for each of the elastic buffers of the example during some cycles, 0 means the EB does not store a new token, 1 means the EB stores a new token, (c) system implemented using scheduling.

using some formal verification tool like NuSMV [44].

It must be verified that all elastic channels comply with the protocol. In the case of SELF, persistence and liveness must be proven. For dual channels, persistence must be verified both for tokens and anti-tokens, and the dual invariant must also be verified (a token is never stopped and canceled). It is also important to prove a correct interaction with the datapath to make sure that no tokens are lost.

In [148], a framework for validating families of latency-insensitive protocols is proposed. The output behavior of the original system and the output behavior of the elasticized system are compared on a subset of possible inputs, allowing quick debugging of new protocols without an exhaustive verification.

It must be ensured that functionality is not lost when transforming a synchronous system into a latency-insensitive one. In [32], it is proven that if all modules of a synchronous system are replaced by the corresponding latency-insensitive modules, the resulting system is patient and latency equivalent to the original one.

Similar results are proven in [96] for synchronous elastic systems. Furthermore, it is shown that when a set of correct elastic systems are composed, they form a new correct elastic system. It is also shown that empty EBs can be added without modifying the functionality of the system.

In [146] a refinement technique is suggested for verifying that an optimized elastic system complies with the original specification. Although it does not apply to systems with early evaluation, it can still be used to prove that an elastic system with bubbles is equivalent to the initial description of the design.

2.2.5 Scheduling

Latency-insensitive designs with only late-evaluation modules tend to have a fixed repetitive behavior [19, 39]. This means that the values for valid and stop bits in the control can actually be predicted a priori. Then, the control logic for each elastic buffer can be reduced to a simple ring of appropriately initialized registers.

Consider the simple example pipeline in Fig. 2.14(a). This pipeline has three elastic buffers connected forming a ring. Initially, EBs *A* and *B* contain a token, while *C* is a bubble. Consider the simulation trace shown in Fig. 2.14(b). In the first clock cycle, *B* and *C* receive a new token. However, *A* does not store a new token because *C* has no token to transmit. On the second cycle, *A* and *C* receive new tokens, while *B* receives a bubble. On the fourth cycle, the configuration is the same as in the first cycle, and from then on the behavior repeats itself. The throughput is 2/3 since every three cycles, two tokens are received by each elastic buffer.

It is possible to synthesize a schedule for each elastic buffer of the design, using some of the algorithms described in [19, 39]. Then, the system can be implemented by using a scheduled control, as shown in Fig. 2.14(c). This implementation does not need any handshake signals. The schedule of each elastic buffer is an automaton the decides, each clock cycle, whether to enable the elastic buffer by following the cyclic sequence described by the schedule.

The main advantage of scheduling latency-insensitive designs is that it removes the need to route the valid and stop wires. These wires may potentially be long and become a problem for routing. However, if the resulting schedule is too long, the area may actually increase as a side effect. Since it is very likely that many of the EBs have the same or shifted schedule, some schedules can be shared to reduce the area overhead [37]. Notice that sharing of schedules can only be applied to EBs that are placed close to each other on the design. In general, systems with early evaluation do not have a repetitive behavior and hence a scheduling cannot be found. If the selection of inputs in each of the early evaluation nodes was predictable and repetitive, then it should still be possible to find a schedule.

2.2.6 Granularity of elastic islands

Elasticity can be applied at different levels of granularity. One option is to view a design as a composition of large synchronous blocks with elastic communication between them in order to use modularity or power advantage of running blocks at different clock frequencies or in anticipation of design variations in communication channels. This block level view using different forms of elastic

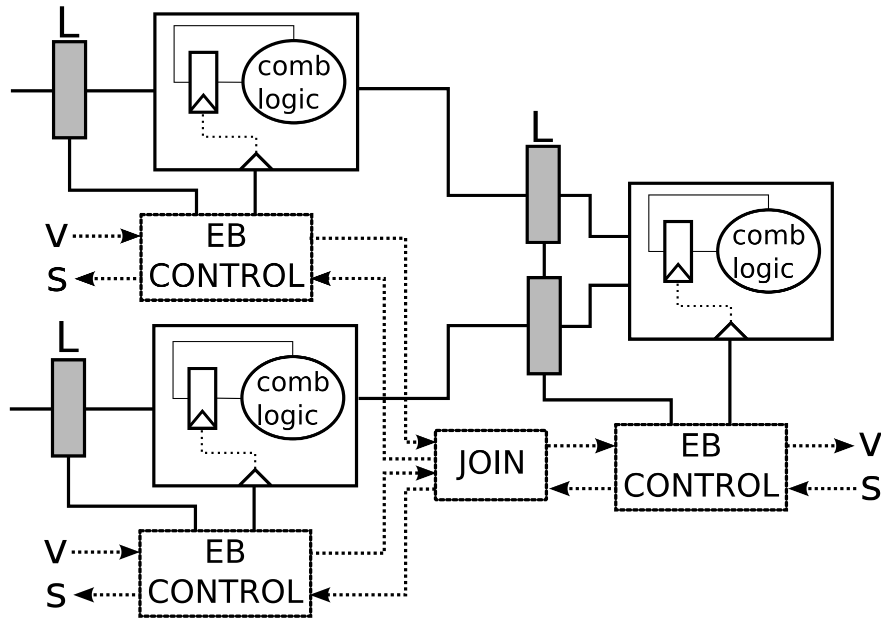


Figure 2.15: Elastic system with 3 IP blocks that have been elasticized by adding an elastic wrapper around them. The control wires are marked with dotted lines.

shells around existing blocks is presented in [34], [152], and [52, §8]. This kind of granularity targets mainly SoC designs.

For example, Fig. 2.15 shows an elastic system where 3 existing synchronous blocks have been elasticized. The global clock of each block is substituted by the gated clock generated by the EB controller. Thus, the elastic controller commands all the registers of the block with only one enable signal. Backpressure is handled by adding *ghost* latches at the inputs of the blocks. These latches have the same polarity as the internal input latches of the block and do not introduce any extra latency, since they are redundant during the normal operation of the system. The ghost latch and the elastic control are the wrapper that must be added to each block to transform it into a pacient shell.

On the other extreme, one can consider a synchronous system as a set of gates. Every gate is then viewed as an elastic island and the system can be redesigned to become elastic at the gate level. Then, each gate has associated handshake signals that communicate with each other in order to properly transmit data. Such low level of granularity would have a prohibitive cost due to the overhead of the huge elastic controller it would require.

An intermediate option is to view the design at the register transfer level and consider every register as an object of interest. Each sequential element at the RTL level is assigned to an elastic

buffer (or relay station) controller, and the synchronization between elastic buffers is extracted by their data-flow dependencies in the RTL design. This view is similar to the one used in the classical retiming optimization, and can be used for elastic systems of the size of a microarchitecture. Even with focus on the RTL level, it is possible to easily group different registers together or decompose them if it is beneficial for performance or area of the design. Using this approach, it is still possible to integrate IP blocks in the design with elastic shells as in the block level approach.

2.2.7 Synthesis Flow

This section describes a synthesis flow that can be used to transform a rigid synchronous system into a synchronous elastic system. This flow is based on the one presented in [36]. Similar flows for synchronous elastic designs have been presented in [31, 53, 82]. It can also be used for *desynchronization*, i.e., to transform a synchronous design into an asynchronous one [20, 54, 93, 151].

This flow can be implemented in an EDA tool and it should be applied during synthesis of the design (either before synthesis on a pre-synthesized RTL or after synthesis on a synthesized netlist). It can also be applied on the initial stages of physical design, but some features like adding bubbles may be less effective on an already placed design. The steps that must be applied are the following ones:

1. Group flip-flops into multibit registers. Each group will be controlled by an EB controller. This grouping (or clustering) determines the level of granularity of the design. Some IP blocks or blocks with a very specific implementation may be kept as is, and they will be elasticized using a coarser granularity. This decision can be taken by a designer or by some algorithm using a set of rules or heuristics. Clear candidates for coarse elasticization are register files or memories.

2. Remove the clock signal and replace each flip-flop with the datapath implementation of an EB (a pair of latches for SELF). In the case of blocks that are elasticized at a coarser granularity, add a wrapper around it to create a shell, i.e., add a ghost latch as shown in Fig. 2.15.

3. Apply transformations to the datapath that will improve the performance of the elastic system, like adding bubbles. All the optimizations that are presented in this thesis can be applied on this stage of the synthesis flow. In particular, early evaluation can be added by identifying early enabling conditions in the datapath and defining which signals must be sent to the controller to recognize such early firings.

4. Create the control layer to generate the enable signals for the latches. It must have the following components:

   (a) An EB controller for each EB (including blocks that have been elasticized at a coarser granularity). The EB controller must be initialized with the correct number of tokens depending on whether it is a bubble or not.

   (b) a Join controller at the input of each EB that receives data from multiple EBs.

   (c) a Fork controller at the output of each EB that sends data to multiple EBs.

   Each join controller might become an early-evaluation join controller. This decision can be taken by a designer or by some algorithm that must decide whether it is wise to apply early evaluation given the datapath and the expected latencies of each of the input branches of the join. At this stage the controller can be synthesized and optimized.

5. Connect the datapath with the controller. The enable signals generated by each EB must be connected to the corresponding items in the datapath, and the datapath signals used by early-evaluation multiplexors must be connected to the early-evaluation joins.

Figure 2.16 illustrates the presented synthesis flow on a simple pipeline example. In Fig. 2.16(b), the flip-flops from the original synchronous design shown in Fig. 2.16(a) have been substituted by pairs of latches in order to implement an EB. The register file in the example (*RF*) has 4 inputs, write address, write data, read address and read data (*wa, wd, ra, rd* respectively). Since register files are complicated blocks where it may not be possible to substitute all flip-flops by latches, they should be elasticized by adding a ghost latch at its inputs.

Figure 2.16(c) shows the whole elastic system after creating the controller and connecting it to the datapath. The multiplexor at the output of the register file has been implemented as an early-evaluation multiplexor. Hence, its select signal is connected to the controller and the join is implemented as an early-evaluation join.

If all paths starting on the read address input of the register file are combinational paths going to the read data output, then its input ghost latch is not necessary. In this case, the implementation shown in Fig. 2.16(d) can be used. The sender of the read address is synchronized with a join at the EB that receives data from the register file. This implementation can be extended to multiple read and write ports and it can be used for memories as well.

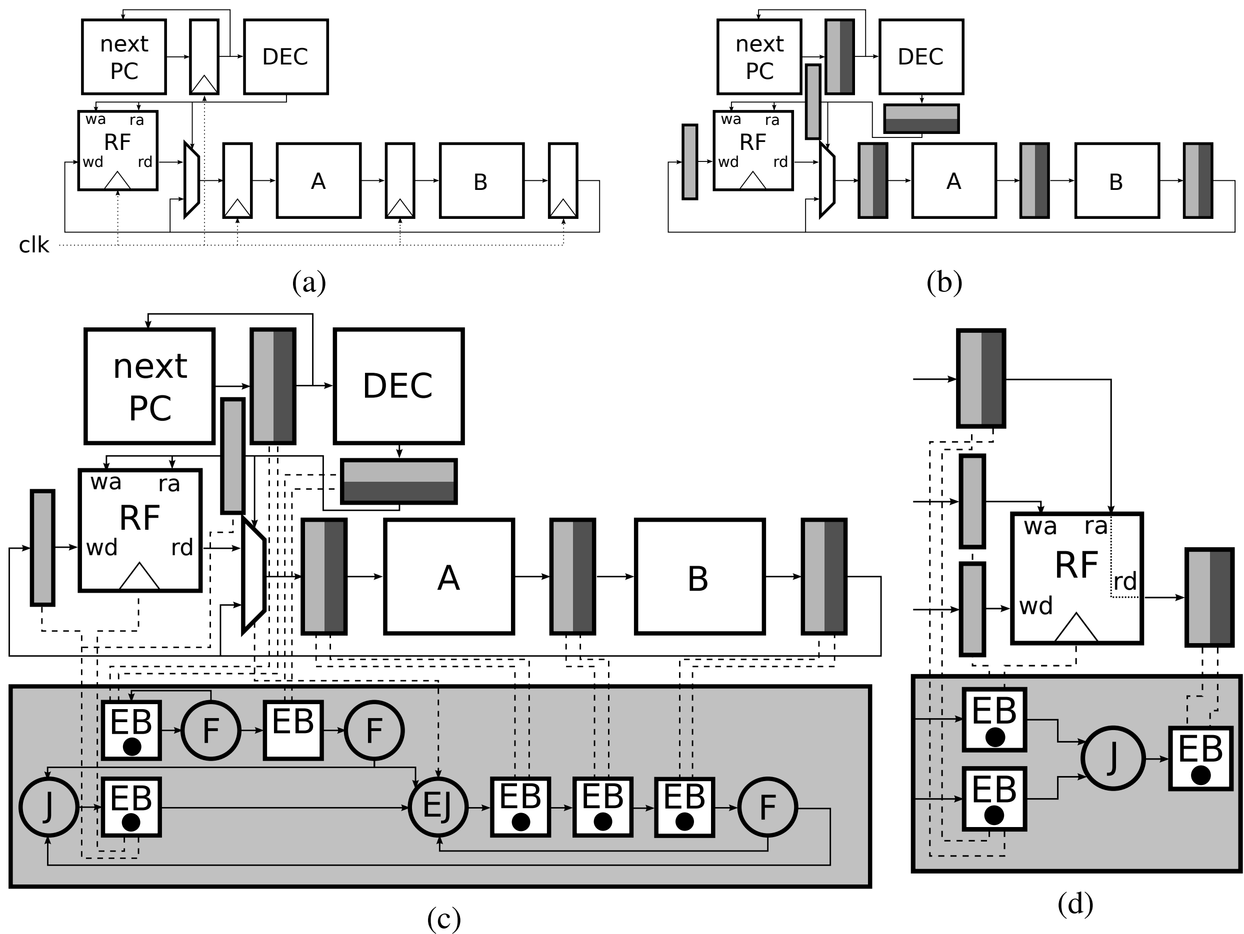


Figure 2.16: (a) Simple synchronous pipeline, (b) Pipeline after converting flip-flops to EBs using a pair of latches, the register file is elasticized using a ghost latch, a bubble has been added after the instruction decoder (c) synchronous elastic system with controller; J means join, F fork and EJ early join; EB controllers are marked with a dot if they initially contain a token; paths from control to datapath and from datapath to control are dashed, (d) alternative way to elasticize the register file if the path from read address to read data is purely combinational.

2.2.8 Modeling Elastic Systems with Marked Graphs

Petri nets can model distributed concurrent computations and their synchronization, and hence, they are an ideal model for both asynchronous systems [112] and synchronous elastic systems [51]. Petri nets are well defined mathematical entities. Therefore, they have been used by several asynchronous groups for performance evaluation, validation, verification, formal reasoning or synthesis of asynchronous systems. Most of these applications can also be used in synchronous elastic sys-

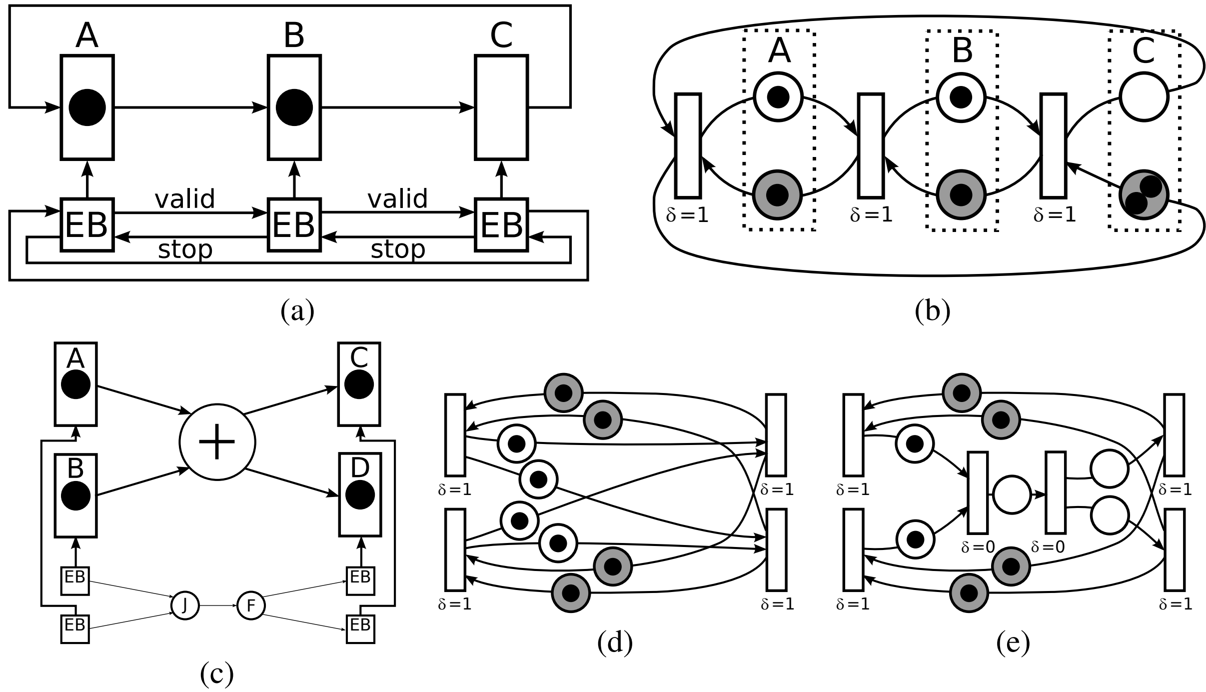


Figure 2.17: (a) Simple elastic pipeline and (b) its elastic marked graph model, each pair of places represent an elastic buffer, the upper place is the forward place and the shadowed lower place is the backward place, which represents available capacity, (c) elastic buffer with joins and forks and (d) its elastic marked graph model, (e) equivalent elastic marked graph with join and fork structure explicitly modeled.

tems.

Since synchronous elastic systems do not have choices, marked graphs have enough expression power to describe them. An Elastic Marked Graph (EMG [51]) is a timed marked graph where the time assigned to transitions is a natural number, since time represents the latency of elastic buffers, which is discrete.

EMGs label each place as a forward place or a backward place ($L : P \longrightarrow \{F, B\}$). Forward places store data tokens that are stored on elastic buffers and are available to be computed by a transition. Backward places store tokens that represent available capacity of elastic buffers that can be filled with new arriving tokens.

Thus, an elastic buffer is modeled with a forward place and a backward place. For every place p, there exists p', such that ${}^\bullet p = p'^\bullet$, ${}^\bullet p' = p^\bullet$ and $L(p) = F$ iff $L(p') = B$. The total capacity of the buffer can be found by adding the number of forward tokens plus the number of backward tokens.

The example in Fig. 2.17(b) has three elastic buffers, A, B and C. In the model, each of them has two places, one in the forward direction and one (shadowed) in the backward direction. Since C is a bubble, initially it has two available spots to receive tokens (the backward place has two tokens) and it contains no tokens in the forward place. A full EB would have two tokens in the forward place and no token in the backward place. The other EBs are half full, and hence they have one token at each place. Since the number of tokens per cycle is a constant in marked graphs, it is ensured that each EB will at most have as many tokens as its capacity. When a data token is transmitted in the forward direction, it can be considered that a bubble (that is, available capacity), is being transmitted in the backward direction. For example, if B sends a token to C in the example, it can also be considered that B obtains a bubble from C.

Figure 2.17(b) shows a linear pipeline. If the elastic system contains join and fork structures like in Fig. 2.17(c), there must be a pair of places (a forward and a backward place) for each EB to EB connection, as shown in Fig. 2.17(d). If the specific join-fork structure must be modeled in the TMG, for example, because one of the joins is an early join or some transition has a variable latency, the backward places must still go from EB to EB, as shown in Fig. 2.17(e).

Early evaluation can be modeled with Multi-guarded marked graphs (GMG, [49, 85]). Multi-guarded MGs allow firing of some transitions even if some of the input places have no tokens. Each early-evaluation transition is assigned a set of guards, which is a subset of its input places. After each firing, the transition chooses non-deterministically one of its guards. Once all the input places of the selected guard have tokens, a new firing can occur. An anti-token is inserted to each input place that was empty when the firing happened. Then, the next guard is selected. Each guard selection is independent.

2.2.9 Performance Analysis of Elastic Systems

The marked graph model of an asynchronous system can be used for performance evaluation. The performance of an asynchronous system is determined by its cycle time, i.e., how often a token is transferred through a transition. Evaluating the throughput of a synchronous elastic system is an analogous problem to computing the cycle time of an asynchronous system. The difference is that in asynchronous designs the time of the transitions model actual time (e.g. picoseconds), while in synchronous elastic systems the transitions model latency in clock cycles.

The asynchronous community has found multiple techniques to evaluate the performance of a system. There are methods based on analysis of Markov processes [97, 156], linear program-

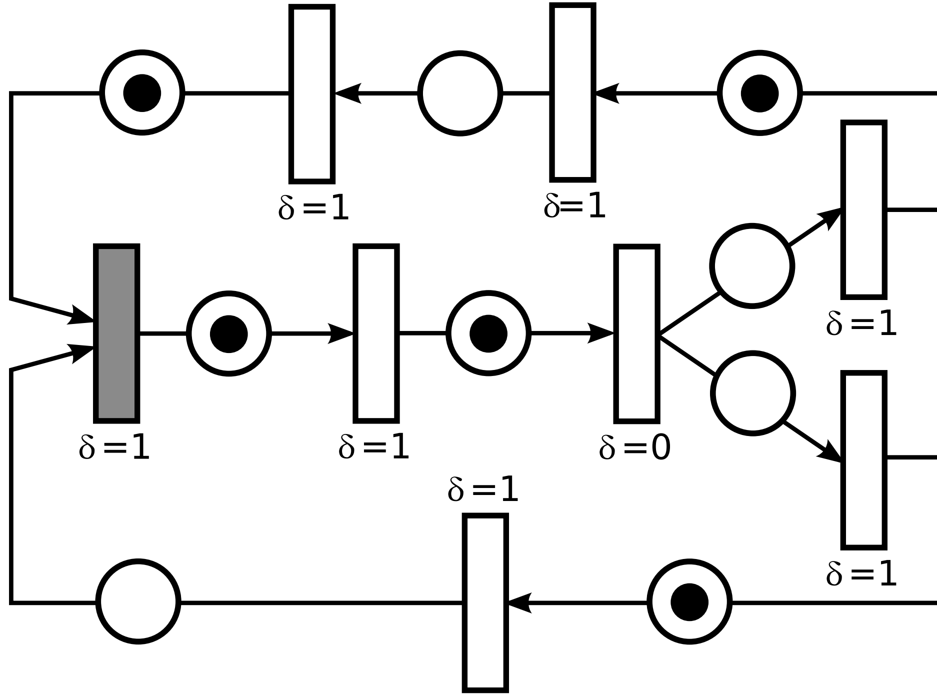


Figure 2.18: TMG model of a synchronous elastic system with two cycles.

ming [30, 46] graph structure of the marked graphs [15, 131, 132, 154], simulation-based techniques [28, 110, 118, 157], partial order between events [42, 80], max-plus algebra [70] or hierarchical methods [67].

One of the most used techniques is based on an analysis of the marked graph structure. Consider a single ring of EBs, as shown in Fig. 2.17(b). Its throughput is determined by its token-delay ratio, i.e., the number of tokens it has divided by the number of clock cycles needed to traverse the ring (the latency of the ring). For example, the throughput of the design in Fig. 2.17(b) is 2/3, since there are 2 tokens and they need 3 clock cycles to return to their starting point. This means that each EB transmits a token 2 of every 3 clock cycles, or that the average separation between consecutive transmissions on the same channel is 1.5 (3/2).

It can be proven that the throughput of a TMG is equal to the throughput of the slowest simple cycle [131, 132]:

$$\Theta = min_{c \in C} \frac{M_0(c)}{\sum_{t \in C} \delta(t)}$$

For example, the TMG in Fig. 2.18 has two simple cycles, the upper one, with a latency of 5 clock cycles and 4 tokens, and the lower one, with a latency of 4 clock cycles and 3 tokens.

The backpressure places have been removed for simplicity. The throughput of the cycles are, hence, 4/5 and 3/4, and the overall throughput is their minimum, 3/4. There are efficient polynomial algorithms that allow to compute the minimum token-delay ratio without enumerating all cycles [57, 89].

The fact that the slowest cycle is the one that determines the throughput can also be translated into a linear programming formulation [30]. In this formulation, the average marking of each place is computed, i.e., how much time a token spends on each place on average. It is known that the throughput multiplied by the delay of a transition must be less or equal than the average marking of all the input places of this transition. Thus, the throughput can be found by solving the following LP problem:

$$\begin{aligned} \text{maximize:} \quad & \Theta, \\ \text{subject to:} \quad & \delta(t) \cdot \Theta \leq \hat{m}(p), \quad \forall t \in T, \quad \forall p \in {}^\bullet t \\ & \hat{m} = m_0 + C \cdot \sigma \end{aligned} \tag{2.4}$$

where $\hat{m}$ is the average marking, which allows real numbers, m_0 is the initial marking, σ is a firing count vector which also allows real values and C the incidence matrix of the MG.

Throughput evaluation of systems with early evaluation is much more complicated. Let us assume that the shadowed transition in the example in Fig. 2.18 is early evaluated, and it chooses either the input from the upper branch or the input from the lower branch. Each guard must be assigned a probability in order to estimate the performance. In practice, it can be assumed that guards are formed by a single input [85]. Hence, each input is assigned a probability. For example, the upper branch is taken with probability α and the lower branch is taken with probability $1 - \alpha$. These probabilities would be determined by profiling the benchmarks expected to be executed on the elastic system.

With early evaluation, the throughput is no longer determined by the slowest cycle. Furthermore, it is not determined by the mean of the throughputs of each cycle even for simple cases. For the example in Fig. 2.18, the simple cycles have throughputs 3/4 and 4/5. However, the overall throughput is not $\alpha \cdot 4/5 + (1-\alpha) \cdot 3/4$, it is $\frac{3+\alpha}{4+\alpha}$, determined by analyzing the *Markov chain* [155] of the system, as explained in [85]. This is the only known method to compute the exact throughput, but it suffers from exponential state explosion. The previous linear programming formulation is adapted in [85, 86] to support early evaluation. It provides an upper bound of the throughput, and it is usually capable of providing a correct order when measuring the performance of different elastic systems, i.e., if pipeline A has better performance than pipeline B, then this method will also find

that A is better than B for most of the cases. The linear programming problem cannot model active anti-tokens.

2.3 Automatic Pipelining

A few automatic and semi-automatic pipelining approaches have been discussed in the literature.

The high-level synthesis community described scheduling and resource sharing algorithms (e.g. [126]) for functional pipelining and software loop pipelining, but static schedules cannot handle dynamic dependencies.

It has also been shown how to automatically pipeline logic blocks without adding latency using a negative/positive register pair [73]. The output of the negative register must either be precomputed or predicted. However, this is not possible within a critical loop (unless an expensive unrolling operation is attempted).

[106] and [95] address the problem of verifying a pipeline via deconstructing it using term-rewriting to derive an equivalent ISA model. [106] applies microarchitectural rules without using precise design on its timing and control. [95] describes a method to add forwarding logic and stall engine to a specification already partitioned into stages. The manual design needs to be proved for correctness afterwards. They use a global controller that needs to send data to all stages. Global controllers that handle stalling and logic forwarding may introduce critical paths in the control of design and are generally not acceptable in the nanometer technologies.

[105] extracts operations from a specification in the form of term-rewriting rules [77], and schedules them into stages interconnected via FIFO queues. Speculation, stalling and forwarding are accomplished by specific styles of rewriting. It is also implemented by using a global controller.

[78] uses term-rewriting rules in order to specify a high-level description of a design, and then the specification is automatically synthesized into a synchronous elastic system, which has a distributed controller.

[76] proposes a framework to specify and verify pipelines where designers follow a template to specify pipeline stages and choose from a library of control cells. As a result, verification scripts about pipeline properties can be generated automatically. Their approach does not use provably-correct transformations and does not generate optimal stalling and bypass based on data dependencies.

[111] proposes a method to explore different possible pipelines starting from an architectural specification. Transformations such as addition of pipeline stages, pipeline paths, opcodes and

new functional units are supported. However, some manual work is required in order to identify and insert forwarding paths. They focus only on instruction set processors, and hence they cannot pipeline arbitrary sequential datapaths.

[121] presents a method to automatically synthesize a pipeline from a transactional datapath specification. The specification executes one instruction (or transaction) at a time. Each module of the specification is then assigned to a stage automatically forming a pipeline. Possible forwarding paths are automatically discovered by the tool, and then the user can choose which forwarding paths to include in the pipeline. The user can also provide value predictors for some modules so that speculative execution can be inserted if necessary. Data hazards that cannot be solved by forwarding or speculation are solved by stalling the pipeline using a handshake mechanism similar to a latency-insensitive design. The final pipeline preserves the original transaction semantics but allows parallelism by executing multiple overlapped transactions across pipeline stages. In [122] this method is extended to allow in-order multi-threading.

2.4 Conclusions

This chapter has given an overview of the notions needed to understand this thesis. Synchronous elastic systems have been presented, and its specification, synthesis, verification and analysis have been discussed. Finally, the chapter also provides a review of automatic pipelining methods.

Chapter 3

Correct-by-construction Transformations

This chapter collects a set of transformations that can be applied to an elastic design. All transformations are correct-by-construction: it is guaranteed that functionality will be preserved through the exploration of different possible microarchitectures. However, the latency of computations and communications may change since the advantages of synchronous elastic systems are used. Applying these transformations to a conventional sequential circuit would be possible, but difficult and error-prone, since it would be necessary to emulate an elastic system with some controller. These transformations can be used for automatic exploration of elastic pipelines.

Most of the transformations have been already used in other contexts. The novelty of this chapter is that it gathers all of them so that they can be used together to perform microarchitectural exploration. First, the microarchitectural graphs used throughout this work and the toolkit that has been built to manipulate them are presented. Then, a set of latency-preserving transformations is introduced. Next, the set of transformations that are only correct for elastic systems is discussed. Finally, verification of such transformations is discussed. The contributions of this chapter have been published in [87] and in [48].

3.1 Introduction

One of the major features of synchronous elastic systems is their tolerance to latency changes. Such tolerance can be used to design systems optimized for the typical case (in terms of data variability) instead of the worst case. For example, an ALU can be redesigned so that it can compute operations with small operands in one clock cycle, while long operations will halt the computation flow and

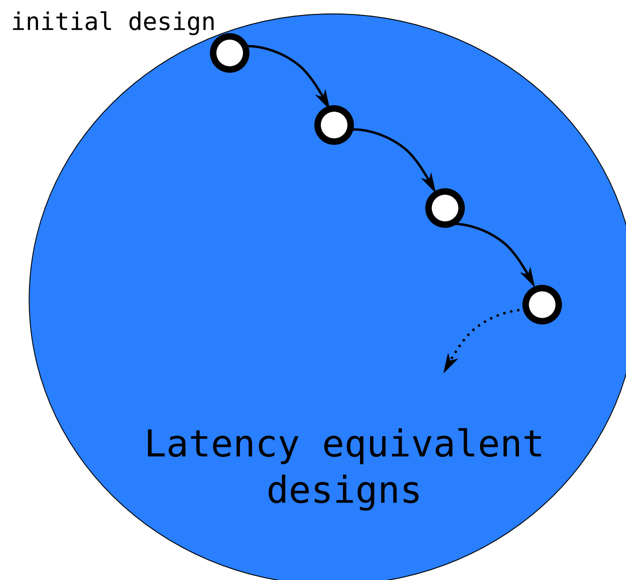


Figure 3.1: Sequence of correct-by-construction transformations applied to a design.

take one extra cycle. Using this variable-latency ALU, the cycle time can be reduced, and assuming the extra cycle is not required too often, an overall performance improvement will be achieved.

An elastic system can be abstracted as a graph where the nodes represent computational blocks and the edges represent channels that transmit data between these blocks. Edges may contain elastic buffers.

Under this abstraction, the transformations that will be presented in this chapter take an elastic graph and modify a small subset of this graph. The resulting graph is latency equivalent to the initial graph, i.e., if idle cycles are ignored, the streams of data on each input and output channel are the same.

By applying a sequence of small steps that perform local transformations to different parts of the design, it is possible to traverse the space of designs that are latency equivalent to the original design, as show in Fig. 3.1. After several transformations, it may not be obvious to determine whether the current design keeps any kind of relation with the original one. However, the fact that all transformations are correct-by-construction and preserve latency equivalence assures that functionality remains unchanged. The optimization of elastic systems by using correct-by-construction transformations can be applied on the third step of the synthesis flow presented in Section 2.2.7.

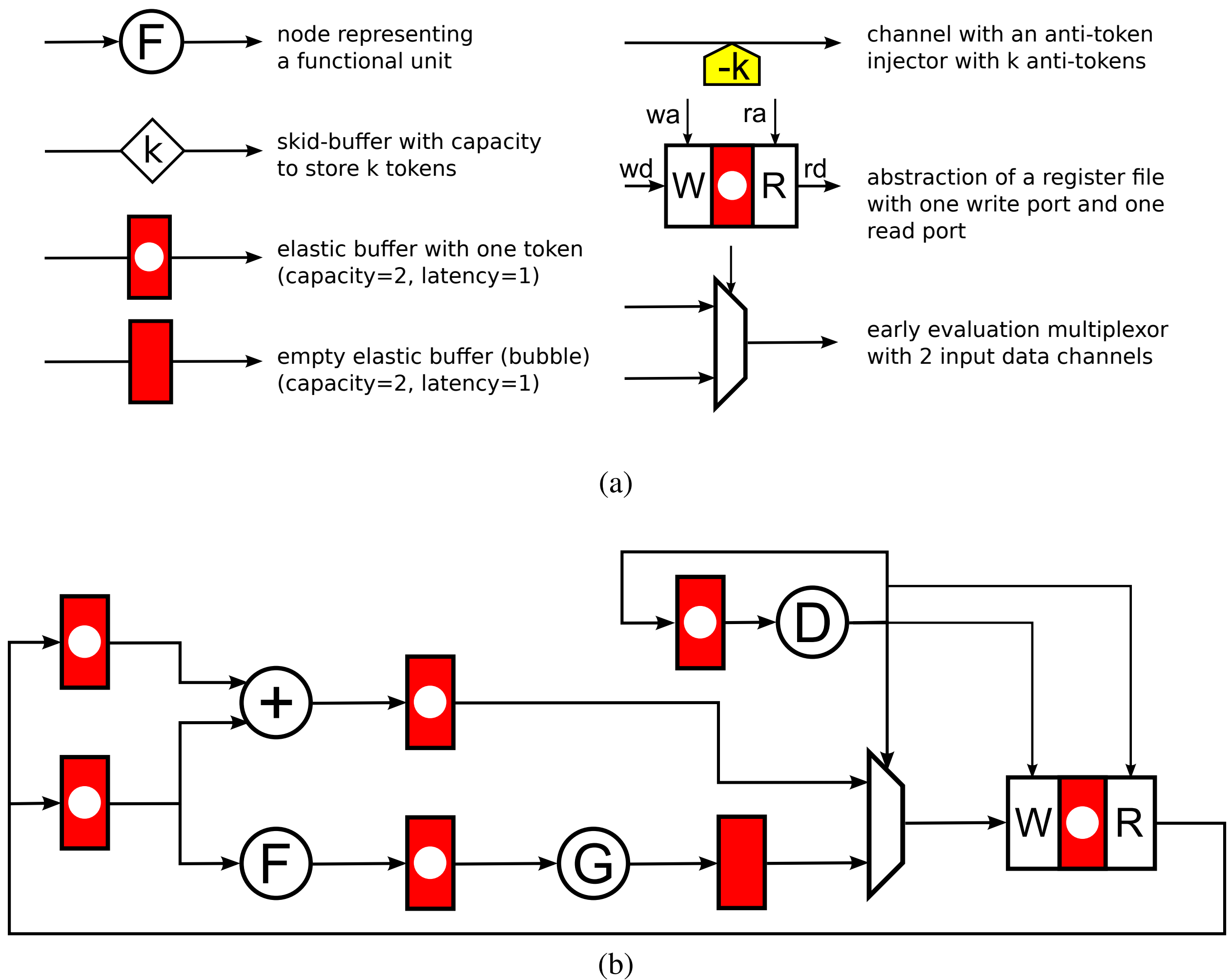


Figure 3.2: (a) Notation to represent elastic microarchitectural graphs, (b) microarchitectural graph example.

3.2 Microarchitectural graphs

Elastic microarchitectures are described in this work using directed multigraphs, which model the Control Data-Flow Graph of the design. A directed multigraph is a directed graph which is permitted to have multiple edges from the same source to the same target. More formally, it is a graph $G = (V, A)$ where V is a set of nodes and A, the arcs of the graph, is a multiset of ordered pairs of nodes.

Microarchitectural graphs can be derived from an actual design, by capturing the graph struc-

ture of the circuit at some level of granularity, or from a functional specification of an Instruction Set Architecture, ISA.

The nodes of the graph correspond to functional units. Each node is assigned an area, a combinational delay and a latency. The latency can be a natural number or a set of natural numbers each one with an associated probability if the node is implemented as a variable-latency unit.

Some nodes may have additional attributes in order to enable early evaluation, sharing of functional nodes, register files, memories, environment nodes and generic IP modules.

The arcs of the multigraph represent elastic channels that transfer data between a sender node and a receiver node. The arcs may contain EBs, skid-buffers and anti-token counters. Each arc is assigned a latency (which determines the number of EBs), a capacity to store tokens, a capacity to store anti-tokens, and an initial number of tokens. The initial number of tokens is an integer which can be negative if the channel initially contains anti-tokens.

Figure 3.2(a) shows some notation used in order to represent elastic microarchitectural graphs in this work. Control details are not explicitly displayed, only the data dependencies are drawn. Elastic buffers are represented as boxes. If the EB is initialized with one token a dot is drawn inside the box. EBs are assumed to have one clock cycle of latency and the ability to store two tokens. Skid-buffers are represented by rhombus, labeled with their capacity. Anti-token injectors are drawn as a pentagon under the elastic channel, labeled with the number of anti-tokens in the anti-token counter. Early-evaluation multiplexors are drawn as a regular multiplexor. At a certain level of abstraction, a register file can be represented by a monolithic register and additional logic to write (*W*) and read (*R*) data. The wires *wd* and *rd* represent data, whereas the wires *wa* and *ra* represent addresses. The actual implementation of the register file within the elastic system would be as discussed in Section 2.2.7.

Figure 3.2(b) shows an example of a microarchitectural graph. There is one register file and one multiplexor selecting between two data inputs. There are four functional nodes, one adding two values and labeled with the plus symbol, two nodes connected in a pipeline and another one which outputs to the select signal of the multiplexor and the write and read addresses of the register file. The data that has not been drawn in this figure are the area, delay and latencies of the nodes, plus the probabilities for each of the inputs of the multiplexor. They are needed in order to evaluate the performance of the design.

The microarchitectural graphs can be hierarchical, i.e., a node may contain another graph inside. The internal contents of a node may be exposed depending on the level of granularity to be applied to the elastic system. For example, a node may represent an ALU, and it may contain an-

other microarchitectural graph with a description of the implementation of the ALU (e.g., an adder and a multiplier connected to an early-evaluation multiplexor). Then, it is possible for the user to decide whether to elasticize the ALU as a whole or to insert transformations inside the ALU to further optimizer the system. Hierarchy allows description of larger graphs.

3.3 Toolkit

A tool named MAREX has been built to describe and manipulate microarchitectural graphs. In order to interact with the tool, the user enters command scripts within an interactive shell, in flavor of other systems with interactive interfaces (e.g. SIS [139] or ABC [1]). Scripts can also be executed in batch mode. The core of MAREX is written in C++ for a performance reason, while Python is used as the front-end for interactive use. In total, MAREX contains around 40K lines of code, developed for the most part by the author of this thesis.

MAREX can apply correct-by-construction transformations on its graphs, visualize the modified graph, undo and redo the transformations, etc. Since most transformations are local, i.e. they are applied on a small sub-graph, they are very fast to compute. Longer scripts that compose a sequence of transformations or implement more complex algorithms can be built as well.

The cycle time of the graph can be computed by traversing the graph using the delays associated to the nodes. It is possible, at any point, to compile the microarchitectural graph into a Verilog netlist of the elastic controller, a marked graph for performance analysis, as explained in [25], a blif model for logic synthesis with SIS [139] or a NuSMV [44] model for verification. Other models for global optimizations like EB capacity sizing or retiming and recycling can also be generated. The elastic controller is built by assembling the set of predefined parameterized control circuit primitives presented in Section 2.2. Verific's front-end library for manipulating Verilog is used to generate the controller RTL. For calculating performance of the design point the Verilog netlist of the elastic controller and the specified model of the datapath is simulated and the throughput is obtained. The throughput can also be estimated by generating a marked graph and analyzing its performance.

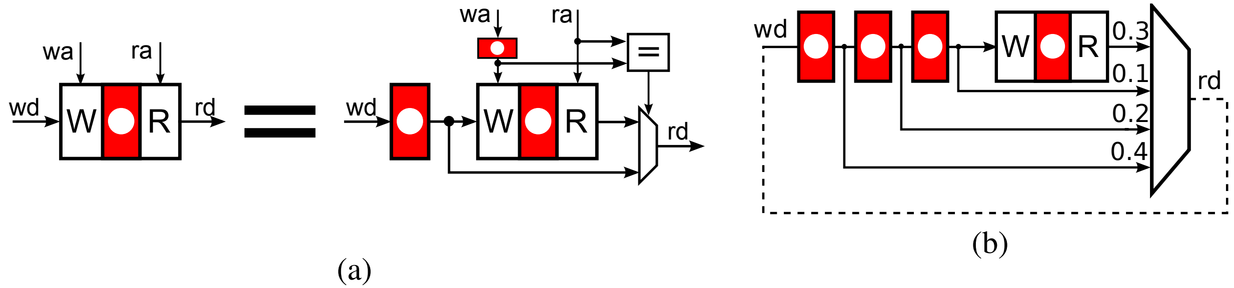


Figure 3.3: (a) Bypass transformation, (b) Register file after three bypasses with probabilities assigned to data inputs.

3.4 Latency-Preserving Transformations

Most design transformations used in conventional synchronous systems can also be applied to elastic systems after some minor modifications. In general, it is correct to map any logic synthesis optimization to the presented microarchitectural graphs, like node duplication. Such optimizations do not change the latency of the computations and preserve the functionality of the design. This section presents bypass, retiming and multiplexor retiming which will be broadly used in this work.

3.4.1 Bypass

Bypasses are widely used to resolve data hazards in processors [75]. A bypass delays the write operation of a register file one clock cycle. A multiplexor is inserted in the read port of the multiplexor to enable forwarding of the delayed data. In the case of elastic systems, the write operation is delayed by inserting an elastic buffer instead of a regular synchronous register. Figure 3.3(a) shows a register file after one bypass. If the read address is equal to the write address of the previous instruction (RAW dependency), the correct data value can be propagated through the forwarding path, even though it has not yet been written in the register file.

The concept of bypass has been used as early as [16] and was leveraged for CAD in [74, 95, 105, 106]. The area overhead of the bypass transformation is one multiplexor, one comparator to detect dependencies by comparing the write and read addresses, and two EBs to delay the write data and the write address.

The multiplexor selecting between the forwarded data and the register file data is typically an early-evaluation multiplexor. Probabilities must be assigned to each of the data inputs in order to enable throughput analysis of the elastic system. Such probabilities should be derived from

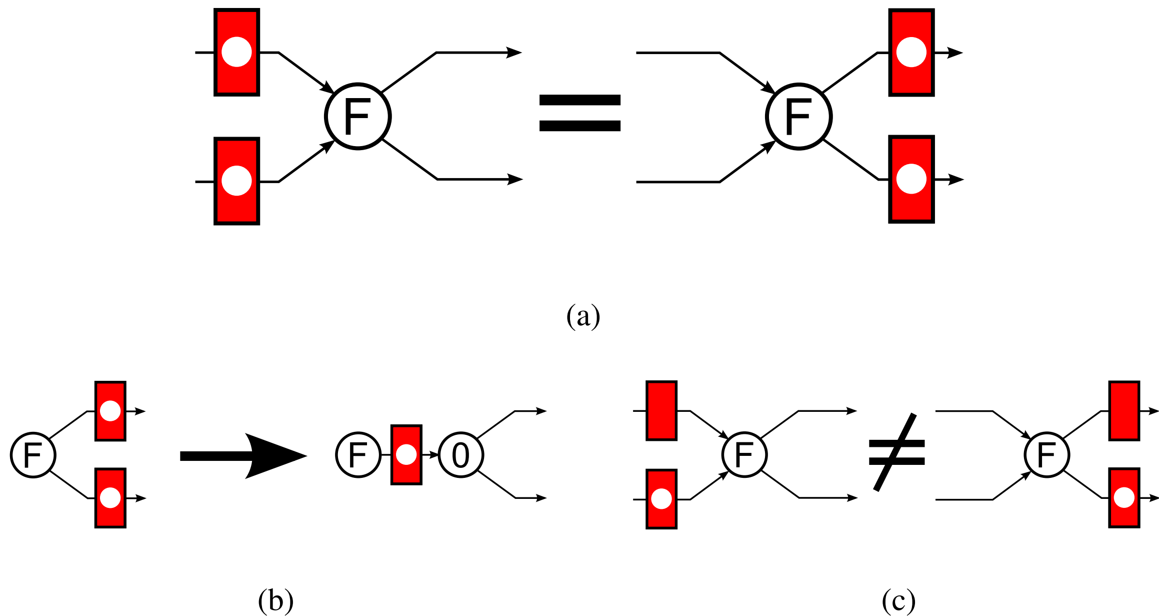


Figure 3.4: (a) Retiming transformation, (b) adding a fake fork to enable sharing of elastic buffers, (c) invalid elastic retiming.

profiling the benchmarks expected to be run on the microarchitecture. It must be determined how often a data dependency can occur for every possible timing distance between instructions, i.e., for instructions that are executed consecutively, for instructions that are executed with one instruction in between, with two instructions in between, and so on. This way, multiple bypasses can be applied and the corresponding probabilities can be assigned to each new early evaluation multiplexor.

Figure 3.3(b) shows a register file with three bypasses and assigned probabilities. The read and write address modules have been ignored for simplicity. The output of the early-evaluation multiplexor may eventually write back to the register file, as shown by the dotted line. In this example, it has been determined that the running instruction will require the value written by the previous instruction with probability 0.4, the value written two instructions ago with probability 0.2 and three instructions ago with probability 0.1. Therefore, there will be no data dependencies with probability 0.3.

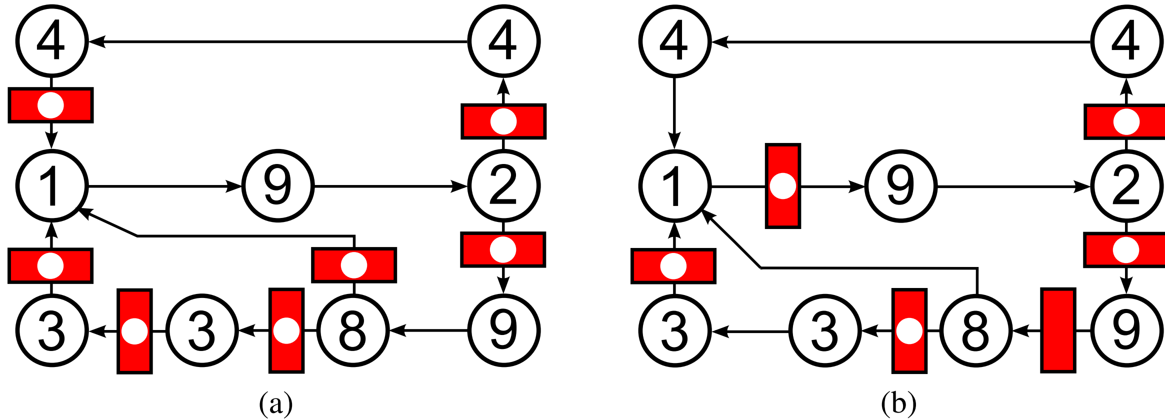


Figure 3.5: Design optimized using (a) Retiming, (b) Retiming and recycling.

3.4.2 Retiming

Registers can be moved across combinational logic preserving the functionality, as shown in Fig. 3.4(a), where a register is moved through node *F*. Retiming [98] is a traditional technique of sequential delay and area optimization. It can also be used to reduce the power consumption of a system [113].

When applying retiming to a synchronous elastic system, it must be ensured that the retimed registers have the same number of initial tokens. For example, the transformation shown in Fig. 3.4(c) is not valid since one of the EBs is a bubble and the other one is not.

The optimization problem of minimizing the cycle time of a design using retiming can be solved in polynomial time [98, 137, 141]. If the target is to minimize the area (i.e., the number of registers) under some timing constraints the problem becomes NP-complete [98], although it may be solved efficiently in some particular cases [81]. The results for area minimization can be improved by adding a fake fork node with zero delay [98, 136], as shown in Fig. 3.4(b). All these methods can be applied to the EBs of synchronous elastic systems. Since the initial state of the circuit may be changed by retiming, a new initial state must be computed after retiming to ensure correct initialization. This is always possible if the retiming graph is strongly connected [150].

Figure 3.5(a) shows an example with an optimal retiming. The combinational nodes (shown as circles) are labeled with their delays. The cycle time of this design is 17 time units, determined by the two nodes with delays 9 and 8 in the lower right corner.

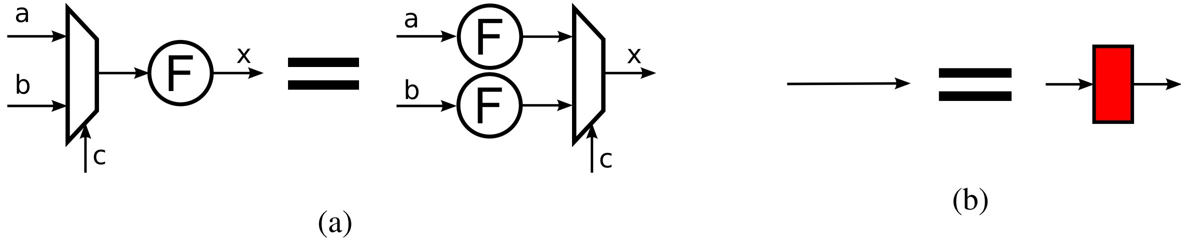


Figure 3.6: (a) Multiplexor retiming and (b) bubble insertion.

3.4.3 Multiplexor Retiming

Functional units can be retimed through multiplexors by adding one copy of the functional unit to each data input of the multiplexor, as shown in Fig. 3.6(a). Such transformation can be used to move logic out of a loop going from the output of a multiplexor to its control channel, similar to what is proposed in [144].

It is easy to show that x in the figure is the same function before and after the transformation, using cofactors and Shannon decomposition:

$$x = F(c'a + cb) \tag{3.1}$$

$$= c'F_{c'}(c'a + cb) + cF_c(c'a + cb) \tag{3.2}$$

$$= c'F(a) + cF(b) \tag{3.3}$$

3.5 Recycling

Recycling is a transformation that changes the latency of an elastic channel. It is always possible to insert and remove an empty EB (a bubble) on any channel of an elastic system (for formal proof see [96]). This transformation is shown in Fig. 3.6(b).

Bubble insertion is also known as recycling, and was initially introduced in [33]. In synchronous elastic designs, recycling can be combined with retiming leading to a more powerful design optimization [26, 35]. Reducing the cycle time of the elastic system by inserting bubbles usually yields to a lower throughput since there are fewer tokens per cycle. By properly balancing cycle time and throughput, the system with the optimal effective cycle time can be achieved.

Figure 3.5(b) shows an optimal configuration combining retiming and recycling for the example from Fig. 3.5(a). The cycle time has been reduced to 11 units. The throughput is determined

Figure 3.7: Capacity sizing in a channel.

by the slowest cycle. The token-to-register ratios for each cycle are 1, 4/5 and 2/3. Therefore, the throughput is 2/3, and thus the average number of cycles to process a token is 3/2. The *effective cycle time* is 16.5 time units ($16.5 = 11 \cdot 3/2$). It means that a new token is processed every 16.5 time units on average - an improvement compared to the 17 units of the optimally retimed design.

The concept of inserting empty buffers for optimizing system performance was long known in asynchronous design [104, 154], and it is called slack matching. In [10, 130], exact algorithms for slack matching on choice-free asynchronous systems were presented. This problem is similar to solving the recycling problem. The main cause of throughput degradation because of bubbles are *unbalanced fork-join paths* [10] (or *reconvergent paths* [39, 45]). When two paths starting from the same fork meet at the same join, it is possible that the fork is often stopped by the join because one path has a smaller latency and must always wait for the slower path. In this case, the paths must be balanced by adding more capacity to store tokens on the fastest path.

3.6 Buffer Capacities

While buffer insertion in Fig. 3.6(b) is formulated for the elastic buffer with capacity two, it also holds for the elastic buffer of any capacity $k \geq 0$. Moreover, if the latency of the buffer is equal to 0 (implementable as a FIFO with a bypass, as shown in Fig. 2.7), the performance of the design as measured by the throughput cannot decrease. The rhombus in the capacity sizing transformation in Fig. 3.7 stands for a skid-buffer with capacity k.

Elastic transformations can modify the capacity of the buffers, leading to a deadlock or to throughput degradation. Consider retiming as an example. In standard synchronous systems retiming moves registers through combinational logic. In elastic systems retiming moves EBs instead of simple registers. Hence, each of such retiming moves also involves moving the capacity to store tokens of the EBs from one channel to another one.

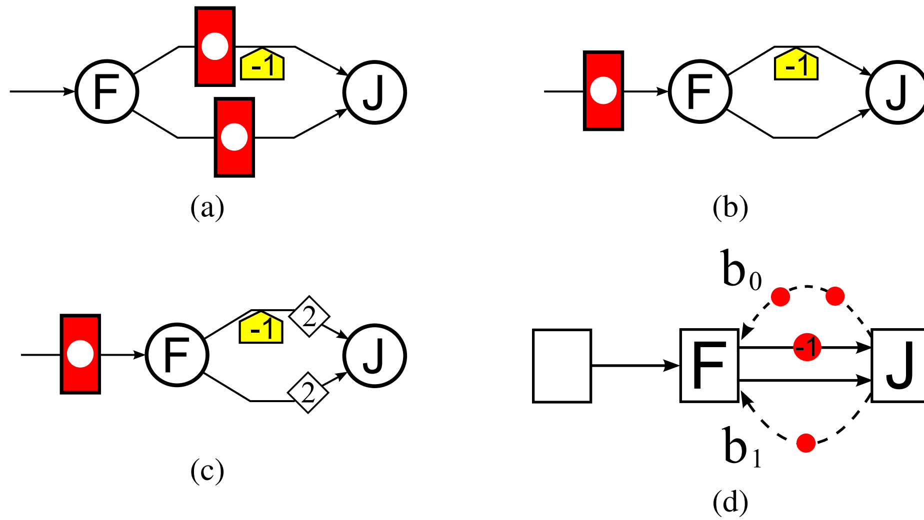


Figure 3.8: (a) Example before retiming, (b) after retiming with deadlock, (c) after retiming with conservative capacity sizing, (d) marked graph that illustrates why the deadlock appears.

Consider a fragment of an elastic system shown in Fig. 3.8(a). After applying backward retiming through node F, one obtains the system in Fig. 3.8(b). This system has a deadlock, as can be seen by the analysis of the corresponding marked graph shown in Fig. 3.8(d)[1]. There are two backpressure edges (shown with dotted lines) going from the multiplexor node to node F. The upper one, b_0, is associated with the upper forward arc. The sum of tokens on the cycle they form is equal to 1 (2 - 1 = 1) which models the fact that node F can fire only once between two firings of the multiplexor node. The lower backpressure arc, b_1, is associated with the lower forward arc, and it has 1 token so that the sum of tokens on this cycle is also 1. However, consider the cycle composed by the upper forward edge and b_1. The sum of tokens on this cycle is zero, a characteristic property of a deadlock in a marked graph.

In order to avoid this deadlock, it is sufficient to add a skid-buffer of capacity one to the lower channel. Then, b_1 would have one more token (with two tokens total) due to the capacity of the skid-buffer. This is a typical example of unbalanced fork-join path, and it can also be fixed by adding a bubble to the lower channel.

Given the throughput obtained with infinite capacities in all buffers, finding the minimal buffer size to reach this throughput is an NP-complete problem [135]. This problem can be solved using integer linear programming [103]. For most of the designs, the number of EBs that need resizing

[1]To be more precise this model corresponds to the dual guarded marked graphs capable of modeling early-evaluation and anti-tokens [86].

is small and a solution can be found quickly. If the design is too big, it is possible to use the polynomial-time heuristic proposed in [45].

Resizing of buffers can be combined with recycling as an optimization procedure [27] that uses Mixed Integer Linear Programming. Combining buffer sizing and recycling can potentially provide better results than recycling alone. Recycling is better in terms of area, but it adds one extra clock cycle of latency. Early evaluation may increase the required size of the buffers for some examples [87].

Instead of solving a costly optimization problem after each retiming move, a skid-buffer with capacity of 2 can be conservatively added to the channels where the EBs were placed before the retiming, as shown in Fig. 3.8(c). This corresponds to adding a skid-buffer of capacity 2 before the retiming move and then performing the move. It can be guaranteed that the resulting system is deadlock free, since the capacity of the channels is never reduced. At the end of the exploration, the optimal buffer sizes can be found by solving a global capacity sizing once.

3.7 Early Evaluation

Introducing early evaluation in some nodes can be considered an optimization technique that allows firing a token even if some of the inputs are not available.

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. 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 [86], and a lower bound method using unfoldings and symbolic expressions is presented in Chapter 6. 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.

Figure 3.9(a) shows the example from Fig. 3.5 after adding early evaluation to the node with delay 1, and adding a bubble on one of the input channels of the multiplexor. Let us assume that the upper channel of the multiplexor is selected with probability 0.7, the middle one with probability 0.2 and the lower one with probability 0.1.

Retiming and recycling can be successfully applied for systems with early evaluation to achieve better performance [24]. The example from Fig. 3.9(a) has a cycle time of 10 time units, which is lower than the 11 units in Fig. 3.5(b). If there were no early evaluation, its throughput would be 0.5, determined by the slowest cycle. Then, the effective cycle time would be 20 units - worse than the

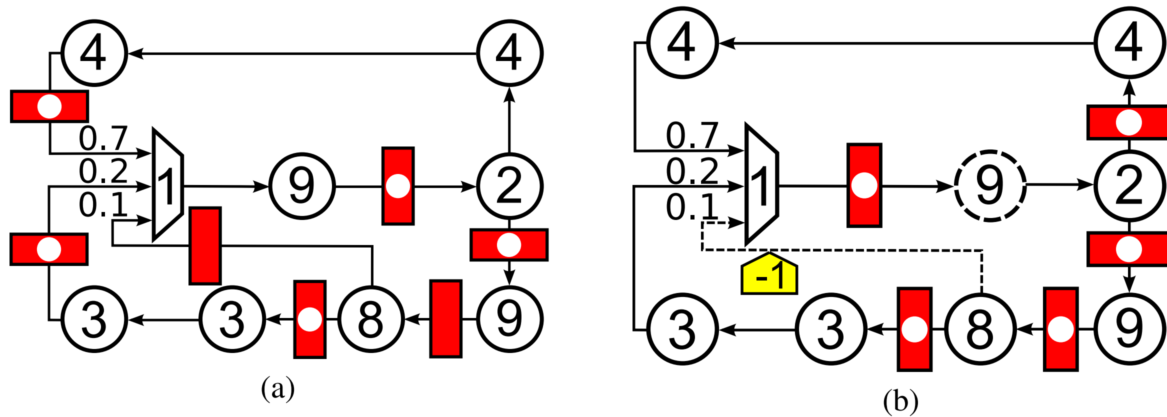


Figure 3.9: Optimal configuration with (a) early evaluation and (b) anti-token insertion.

16.5 units obtained for the previous configuration without early evaluation. However, when early evaluation is introduced, the cycle with the worse throughput is only selected by the multiplexor in 10% of the cases. If the system is simulated using the given probabilities, the obtained throughput is 0.79. Thus, in this example, early evaluation allows to reduce the effective cycle time to 12.65 units (10/0.79).

When there are no early-evaluation nodes, retiming usually provides the optimal solution and recycling does no help much [26]. The throughput is the worst token-to-cycle ratio and hence, adding a bubble is not possible in many places without a big reduction of the throughput. On the other hand, early evaluation allows to design for the typical case in terms of data variability. Data that is not selected often in some of the multiplexors of the design 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.

3.7.1 Critical Cores and Little Cares

Optimization for the typical case can also be performed using little cares [68]. Consider the design with two pipeline stages in a loop shown in Fig. 3.10(a). Assume that both functional blocks, F and G, have a delay of 100 units. Therefore, the cycle time of the system is 100 units and the effective cycle time is also 100 units, since the throughput is 1.

Let us assume that there are two types of flows (e.g. two different classes of instructions) through this simple system: the first one occurs with a high probability, α, and the other one with

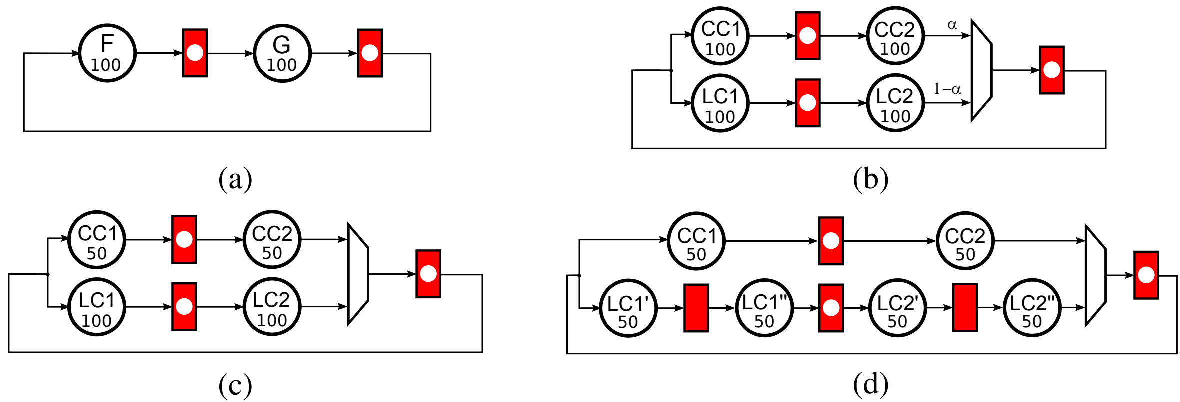


Figure 3.10: Using little cares to optimize a design [68].

lower probability, $1 - \alpha$. The flows that occur rarely are called *little cares* (denoted LC) and the flows that are computed often are the *critical core* (denoted CC). Based on profiling of the available design models (e.g. functional model, performance model or RTL) the CC and LC portions can be identified, split, and multiplexed as shown in Fig. 3.10(b). At this stage timing is still the same.

On Fig. 3.10(c), the critical core portion has been optimized, and its delay has been reduced from 100 to 50 units per stage. Some techniques that are not feasible for the whole design can be applied on the CC, since it is smaller and typically much simpler (see [68] for an example with an industrial video decoder). The overall system performance has not changed in Fig. 3.10(c) since potential timing critical paths going through the infrequent LC portion of the design may still have the original delay of 100 units.

At the final stage of the optimization, shown in Fig. 3.10(d), recycling is applied into timing critical paths of the infrequent computation, LC. Note that this transformation, which may affect a large set of corner cases, is very simple. All stages of the pipeline have a delay of 50 units, and hence the whole synchronous system can run with a clock cycle of 50 units.

If the CC is taken alone, the speed-up is 2x. However, if the LC is taken alone, the performance has not changed, since the cycle time has been reduced to half but the throughput has also been reduced to half. The overall performance depends on how often CC is computed and how often LC is computed. For example, if $\alpha = 0.9$, then the average effective cycle time is 60 units, which is 1.67x improvement from the original design.

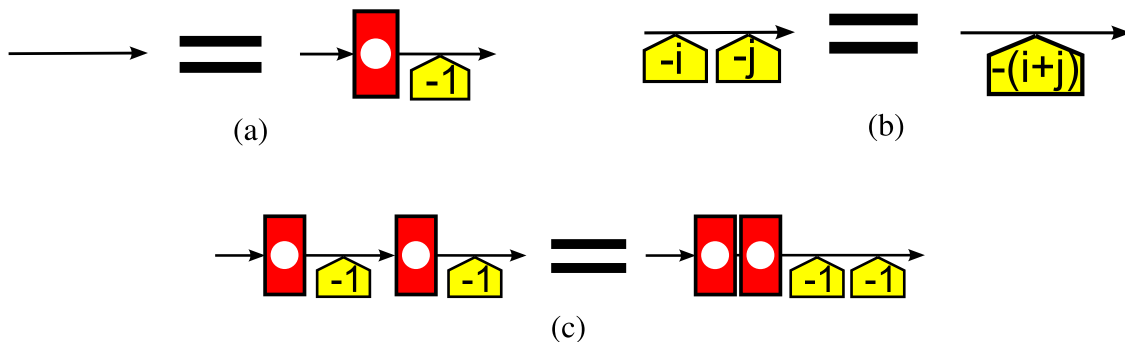


Figure 3.11: (a) Anti-token insertion, (b) anti-token grouping, (c) anti-token retiming.

3.8 Anti-token Insertion

Anti-tokens are used to cancel spurious computations in early-evaluation nodes, but they can also be used to enable new retiming configurations. 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. 3.11(a). The anti-token injector can be implemented by a modulo-2 up-down counter placed on the elastic channel (without changing the latency of the channel). When a token flows through a non-empty anti-token injector, the token and the anti-token cancel each other. The anti-token simultaneously disappears (the counter is decremented).

Anti-token counters can be retimed (as in Fig. 3.11(c)) and grouped (as in Fig. 3.11(b)). When retiming anti-tokens, care must be taken with the initial values of the registers so that functionality does not change. Furthermore, 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 through the node.

Anti-token insertion can be often applied to enable retiming of EBs that have been initialized with a different number of tokens. For example, Fig. 3.9(b) shows a system where 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 its throughput is very high, 0.918, since there is only one cycle with a bubble (a sum of a token and an anti-token is equal to zero) as compared to Fig. 3.9(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.

3.8.1 Passive or Active Anti-tokens

Let us consider the differences in using active and passive anti-tokens. In most of the cases, the throughput provided by passive anti-tokens is the same as the throughput provided by active anti-tokens. Therefore, since passive anti-tokens are more efficient in area (no need to create a dual control, route twice as wires, etc), they should be the preferred choice when implementing early evaluation or anti-token injectors.

However, in some examples active anti-tokens provide a higher throughput, as noted in [41]. Let us explain why. Consider the example in Fig. 3.12(a). There are 6 EBs, labeled from EB_0 to EB_5, and two early-evaluation multiplexors, EE_0 and EE_1. The output of EE_0 is connected to the output of EE_1. The tokens in the buffers that are connected to the select signals of the multiplexors are labeled with their data values, while the rest of the tokens, initially in EB_0 and EB_1, are labeled with different letters so that they can be easily tracked. F_1, F_2 and F' are three functions that are applied to the inputs coming from EB_0.

Let us simulate this example cycle by cycle with active anti-tokens. In the first cycle, shown in Fig. 3.12(b), EE_1 chooses the lower channel driven by EB_1. Hence, token *D* is transmitted to EB_3. An anti-token is generated by EE_1, and it travels backwards. It cancels the token in EB_4, it arrives to EB_2 where it will cancel token *A* inside the EB [2], and it also cancels token *A* going through the lower branch of the output fork of EB_0.

During the second cycle, shown in Fig. 3.12(c), EE_0 selects the lower channel and EE_1 selects the upper one. Therefore, token *B* can be transmitted directly to EB_3. Token *B* is also sent to EB_2, where it will be canceled by the anti-token generated at EE_0. Finally, token *E* is transmitted to EE_1 where it will be discarded because it has not been selected. After two cycles, the system in Fig. 3.12(d) is achieved, where tokens *D* and *B* have been sent to EB_3.

Consider now the same simulation with passive anti-tokens. In the first cycle, shown in Fig. 3.12(e), token *D* is sent to EB_3. One interesting detail is that token *A* is sent to EB_2 but at the same time it is stopped on the lower branch, because EE_0 must wait for EB_2. Since the fork is an eager fork it can still send *A* through the upper channel. This is a so called unbalanced fork-join path (see Section 3.5). A skid-buffer on the lower channel would allow EB_0 to completely send *A* and move to the next evaluation. This is not significant for the performance of the steps that are shown in this example, but it would be significant for the overall throughput of the system.

[2] EB_2 initially stores token *A* because from the input interface it cannot know that an anti-token arrived at the output interface. Inside the EB_2 controller, the token and anti-token will meet and nullify each other. Hence, token *A* will never reach the output of EB_2.

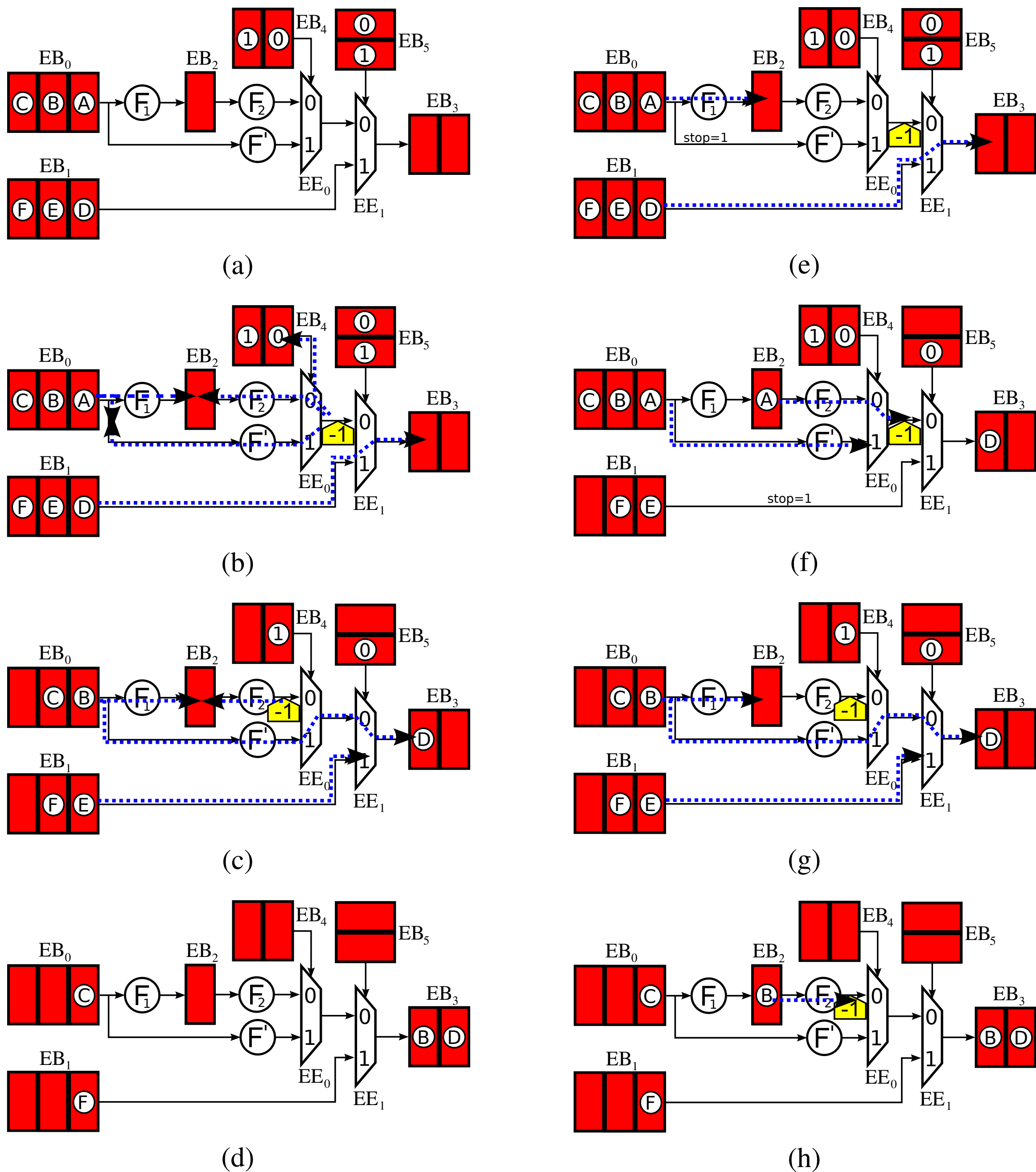


Figure 3.12: (a) Example design with back-to-back early-evaluation multiplexors, with difference performance depending on whether the anti-tokens are passive or active, (b),(c),(d) cycle by cycle simulation with active anti-tokens, (e),(f),(g),(h) cycle by cycle simulation with passive anti-tokens.

An anti-token is generated at the upper input of EE_1. It is a passive anti-token. Hence, it sits and waits for tokens to arrive inside the early join controller. Notice that if the multiplexor keeps selecting the lower branch, the token flow will not be interrupted at all.

In the second cycle, shown in Fig. 3.12(f), token *A* has arrived at the output of EB_2 and hence EE_0 is able to fire it. When *A* reaches the anti-token in EE_1, it is canceled. Meanwhile, EE_1 has selected the upper channel, and hence, token *E* has been stopped and it is waiting for token *B*. During the third cycle, shown in Fig. 3.12(g), token *B* can finally be sent to EB_3, and a passive anti-token is generated in the upper channel of EE_0 waiting for EB_2 to be valid.

Finally, in Fig. 3.12(h) the equivalent marking of Fig. 3.12(d) has been reached. Using passive anti-tokens provides one clock cycle overhead for this particular trace. In [23, Ch. 5, S. 5], a very similar example is formally analyzed where active anti-tokens provide a 7% improvement in throughput. In this simple example, it would be possible to just merge the two early joins into a single early join with three inputs in the controller, and there would be no difference. If the multiplexors were separated by one EB, merging them would no longer be possible, but the performance improvement provided by active anti-tokens would be smaller as well.

In conclusion, active anti-tokens may be more suitable when there are concurrent decisions that affect each other, and they are taken in different parts of the design that are close enough in terms of clock cycles. In these cases, anti-tokens can communicate these decisions faster avoiding unnecessary stalls, as shown by the example in this section. In any other case, passive anti-tokens provide the same performance with less area overhead.

3.9 Variable-latency Units

Variable-latency units can be handled in a natural way in synchronous elastic systems. A handshake with the datapath unit is required so that the control can keep track of the status of operation, as shown in [53].

For example, an ALU may spend one clock cycle to compute frequent operations with small operands (i.e. operands with few significant digits), and two clock cycles for rare operations involving larger operands. This is a typical example of telescopic unit [12, 147]. Variable-latency units can improve the performance by decreasing the overall cycle time and the average time needed per computation. It is possible to automatically synthesize a variable-latency unit out of a regular combinational block minimizing the resulting cycle time [9].

Other examples of variable-latency units are large register files with access time ranging from

one to two cycles for different partitions, variable hit time caches (so called pseudo-associative caches), video decoding blocks processing different video symbols with largely varying probabilities and any other operation where there is a significant discrepancy between the typical and the worst case pattern of operation.

In the example from Fig. 3.9(b), the critical cycle is determined by the dashed node with delay 9 followed by the node with delay 2. Assume the dashed node can be replaced by a variable-latency node that has a typical delay of 7 time units at the cost of spending an extra cycle (i.e. 14 time units) in rare cases. Then, the cycle time of the system will drop from 11 to 9 units.

Let us assume that the short operation can be applied 95% of the times. Then, the throughput of the system is 0.881, estimated by simulating the controller. The resulting effective cycle time is 10.216 units (9/0.881), compared to the previous 11.98. Overall, correct-by-construction transformations that modify latency have provided a 66% improvement in performance for the example shown throughout the chapter.

Notice that a multi-cycle operation must be aborted when an anti-token reaches its controller. Hence, when a variable-latency unit is sending tokens to an early-evaluation module, the throughput of the system can be higher if the anti-tokens are active, since they will preempt unneeded long computations.

3.10 Sharing of Functional Units

When a system includes early evaluation, some computations are not always required, and hence, they can be delayed for some cycles with no performance penalty or even canceled. As a result, the actual utilization of some units can be way below 100%.

Different modules with the same behavior (for example, two adders in the design), can be merged into a single module, which is then shared by the input channels that compete for this resource. The module sharing transformation may provide a reduction of area and power in the design, hopefully at a low (or even zero) performance degradation. It can also be used to introduce speculative execution, as shown in Chapter 4.

For example, Fig. 3.13(a) shows two channels, each one using a node called F, which compute exactly the same function. If both F are shared into a single physical entity, the logical view remains the same, although the latency of F becomes variable.

Every clock cycle, a scheduler selects which channel can use F, as shown in Fig. 3.13(b). The performance variation compared to using unshared resources depends on whether the scheduler

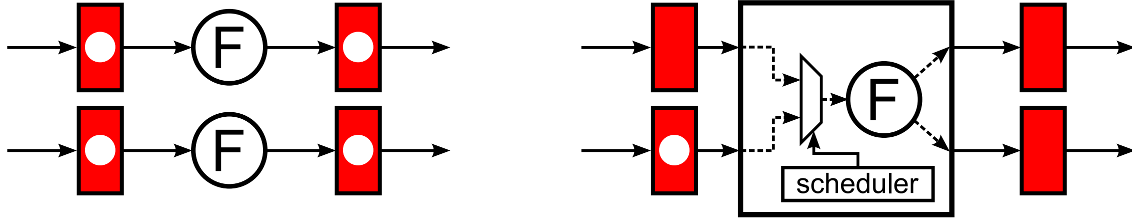


Figure 3.13: (a) Logical view of a shared unit, F is considered a variable-latency unit, (b) physical view of a shared unit, the scheduler of the shared unit grants access to one of the channels.

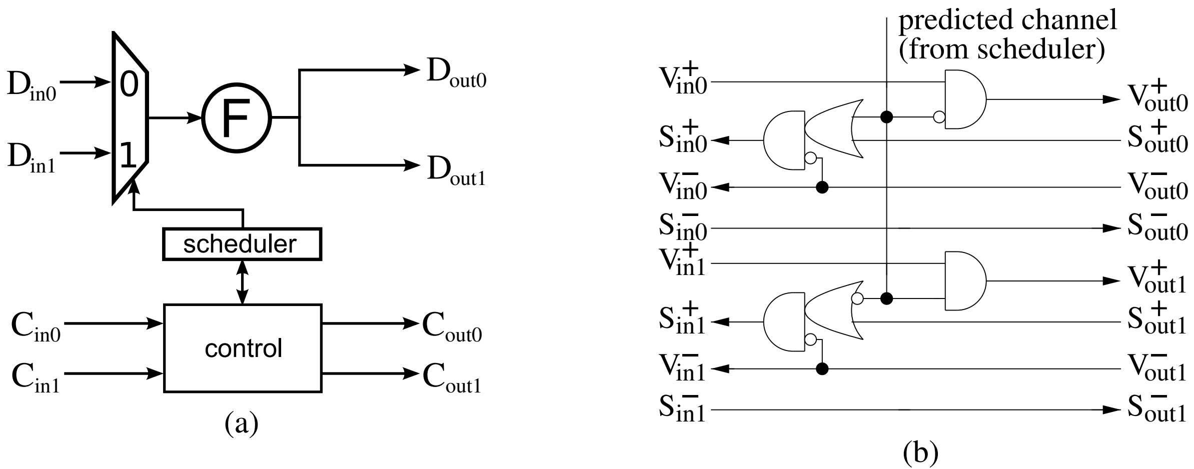


Figure 3.14: (a) Datapath implementation of a shared module, (b) controller of the shared module.

can distribute the load accurately among the different users.

Figure 3.14(a) shows the datapath logic for a combinational block shared by two channels, and Fig. 3.14(b) shows its control logic. In Fig. 3.14(a), C_{in_i} and C_{out_i} represent the handshake control bits of the elastic channels ($C_{in_i} = \{V^+_{in_i}, S^+_{in_i}, V^-_{in_i}, S^-_{in_i}\}$, $C_{out_i} = \{V^+_{out_i}, S^+_{out_i}, V^-_{out_i}, S^-_{out_i}\}$); and D_{in_i} and D_{out_i} represent the datapath wires associated with these control bits. The delay overhead added to the datapath is one multiplexor plus the delay in the scheduling decision. One should make sure that the scheduler is out of the critical path.

The controller sets the valid signal of the selected channel as long as its input is valid and keeps the valid signal of the other channel at 0. It also stops the channel that is not selected unless it is propagating an anti-token. The implementation of the controller can be trivially extended to handle more than two channels.

It is assumed that the shared module is a combinational circuit, i.e., it does not depend on the

results of previous cycles. Otherwise, it would be necessary to replicate the state of the module for each channel that uses it, and make sure that each clock cycle, the state is read and written correctly according to the scheduler.

The scheduler can implement prediction algorithms of different complexity, from a round-robin strategy to more advanced prediction algorithms as long as they do not introduce new critical paths. For better performance, the scheduler should take into account the elastic protocol: an invalid or a stalled channel cannot use the shared unit even if selected.

Using a shared module is like using a module followed by a buffer with unbounded but finite latency, since each data token may have to wait for a certain number of cycles until it is allowed to use the shared module. For correctness, the scheduler should be fair in order to avoid starvation of the channels: every token that reaches the shared module must eventually be allowed to use it unless it is canceled by an anti-token.

This property can be formalized as a leads-to constraint: every arrived token must be eventually served by the shared unit or killed. Formally, for every channel i that uses the shared unit:

$$\mathsf{G}\ (V^+_{in_i} \Rightarrow \mathsf{F}\ (V^-_{out_i} \vee (sel = i \wedge \overline{S^+_{out_i}}))) \qquad \text{(leads-to scheduler property)}$$

3.11 Verification

One can verify correctness of the previous correct-by-construction transformations using model checking, with tools such as SMV [108]. Given the subgraph of the system where the transformation has been applied, it must be verified that the original subgraph is *transfer equivalent* to the transformed subgraph. This can be proven by checking that each channel complies with the handshake protocol before and after the transformation, data token order is preserved and the symbolic function computed by the datapath is preserved at the outputs of the system.

For module sharing, the absence of deadlocks was verified for *any scheduler* that satisfies the leads-to scheduler property. To prove that the fairness of the schedulers is a sufficient condition for liveness, the refinement verification was applied. It was proven that a shared module sequentially composed with an abstract model of an `EB` that has an undetermined finite response latency, as presented in Fig. 2.11, is a refinement of the `EB` specification itself, provided that the shared module has a non-deterministic fair scheduler. That is, sharing adds an undetermined but finite latency to the computation, but it does not change the functionality of the design.

Notice that after a retry cycle on an output channel of a shared module, the scheduler (specified

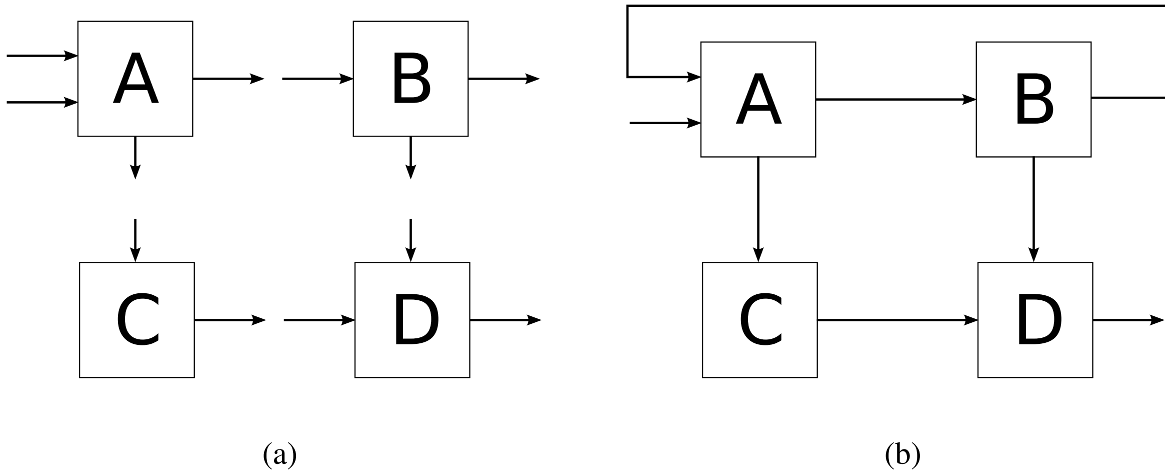


Figure 3.15: (a) Set of elastic modules (b) connected creating a new elastic module.

by the leads-to constraints) is allowed to change its prediction. Hence, the output channels of the shared modules are not required to be persistent. However, persistence is maintained at the inputs of the shared module and at the outputs of all EBs after the shared module, which is enough to ensure that no tokens are lost or reordered.

In [96] it is proven that bubble insertion is correct. It is also proven that an elastic module composed with another elastic module form a new elastic module, even if a feedback loop is added (as long as there are no combinational loops in the design, there is at least one token in the loop and assuming that buffer capacities are big enough). Therefore, given a set of elastic modules that comply with the elastic protocol and that are live, such as the ones shown in Fig. 3.15(a), any possible connection, like the one in Fig. 3.15(b), will form a new elastic module that will still comply with the elastic protocol and will still be live. This also means that if any of the modules in this figure is replaced by another module that is latency equivalent to the initial one (for example, A is substituted by A'), the functionality of the overall system will not change.

This compositional property has only been proven for elastic systems without anti-token propagation. When anti-tokens can be propagated backwards through elastic channels, the transfer traces of the channels are not the same as they were without anti-token propagation, and therefore, the definition of latency equivalence becomes more challenging.

However, compositionality can still be applied where early evaluation and token counterflow are encapsulated within an elastic sub-module that has a regular valid/stop handshake interface, that is, it does not propagate or receive anti-tokens from its neighbors. The presented set of trans-

formations only use anti-tokens locally, and they very rarely need to traverse more than one or two levels of elastic buffers. Therefore, such encapsulation is always possible within a small subgraph where model checking can be used to prove the correctness of the transformation. Then, the transformed subgraph can replace the original one and the global functionality of the design will not change.

3.12 Conclusion

This chapter has presented a collection of correct-by-construction transformations which can be used to modify the microarchitecture of an elastic system. This way, it is possible to explore different systems while it is guaranteed that functionality is preserved throughout the process.

Bypassing, retiming and multiplexor retiming are transformations valid for conventional sequential circuits that have been adapted to elastic systems. The rest of the presented transformations can modify the latency of the computations and hence, they can only be applied to latency-insensitive designs or to asynchronous designs. The verification of the presented transformations has been discussed.

Chapter 4

Speculation

Speculation is a well-known technique for increasing parallelism of microprocessor pipelines and hence their performance. When the outcome of an operation is unknown during some cycle, but is required to perform another operation, two schemes can be considered for a correct behavior: (1) stall the pipeline until the first operation can provide the required data, or (2) *predict* the outcome of the operation and continue the computations without stalling the pipeline. In the second case, the predicted result must be checked for correctness after the first operation has completed and, in case of *misprediction*, the speculated computations must be invalidated. If the predictions are highly accurate, speculation may potentially provide a tangible performance improvement.

The most typical example of speculation is in the execution of branch instructions when the target address is predicted without knowing the outcome of the branch. If a misprediction is detected once the branch condition is calculated, the speculated instructions are invalidated and the correct flow is started.

Speculative execution can be introduced in elastic systems by applying a sequence correct-by-construction transformations, taking advantage of early-evaluation multiplexors and sharing of functional units.

The contribution presented in this chapter is based on the results published in [64]. The introduction of this chapter provides an overview of the proposed method. A more thorough description of the used transformations and the behavior of an speculative execution follows. Next, an optimization in elastic buffer implementation is presented, which allows to improve the throughput of speculative elastic systems by speeding up the transmission of critical anti-tokens. Finally, the benefits of speculative execution in elastic systems are illustrated with two design examples.

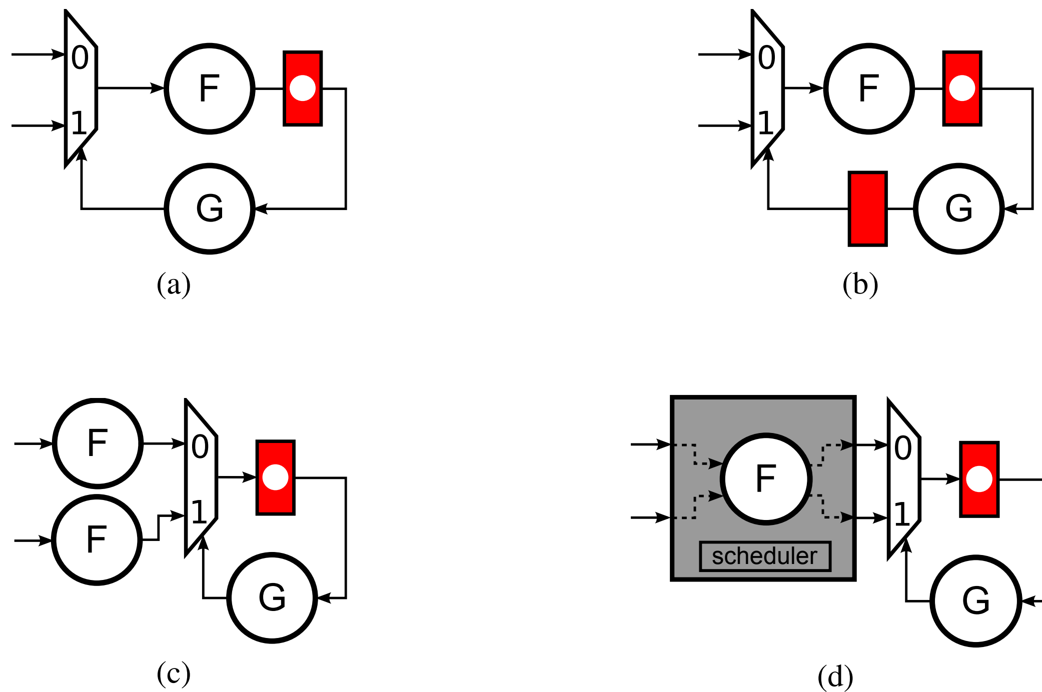


Figure 4.1: Example of speculation in elastic systems: (a) non-speculative system, (b) system after bubble insertion, (c) system after Shannon decomposition, (d) final design with speculation.

4.1 Introduction

The need for speculation arises when there is a decision point in the datapath in which some of the required data arrive late. Figure 4.1(a) shows a simple elastic circuit in which speculation can boost up its performance.

The loop through F and G could represent the computation needed to decide whether a branch instruction is taken on a microprocessor. The two input channels would send the next Program Counter (PC) in case no branch is taken, and the next PC in case the branch is taken. Let us assume that there is a critical path starting at the EB, going through G, the multiplexor, F and arriving at the EB again. A possible way to optimize the performance of this design would be to insert an empty EB in the critical path, as shown in Fig. 4.1(b). While this transformation would improve the cycle time of the design, the throughput of the system would decrease to $1/2$ because there is a loop with one token and two cycles of latency. The token would reach the multiplexor once every two cycles, and during the other cycle there would be no real computation. Therefore, bubble insertion brings no real gain in this example. Retiming is not useful either. The number of

tokens in a loop is constant and cannot be changed.

Given a multiplexor with several inputs, it is possible to move a functional block from the output of the multiplexor to its inputs using Shannon decomposition [144] (viewed also as a multiplexor retiming), as shown in Fig. 4.1(c).

After moving F from the output of the multiplexor to its inputs, F and G are executed in parallel rather than sequentially, achieving a better cycle time. Furthermore, the throughput of the system is optimal as there are no bubbles. However, this speed-up comes at a price: duplication of logic. Here is where speculation can be effectively used. In order to reduce area, all copies of F can be merged into a single one as shown in Fig. 4.1(d). Then, the system must *speculate* which input channel of the multiplexor should first use the shared functional module. The scheduler inside the shared module is the one that must select which input channel can propagate a token through the shared module, thus implicitly predicting the value of the select signal of the multiplexor.

When the speculation is correct, the early-evaluation multiplexor receives the required data and computes its output. At the same time, an *active anti-token* is propagated backwards through the channel that was not selected, invalidating the unneeded data. When it is not correct, the multiplexor stalls. Then, the correct token must be propagated through F.

The design in Fig. 4.1(c) is optimal in performance. However, if the prediction strategy in Fig. 4.1(d) is sufficiently accurate, the cycle clock penalty due to speculation will be rarely paid, achieving a similar performance with smaller power and area.

A manual implementation of all the stalling/canceling mechanisms for speculation is complicated and error-prone. A set of verified control primitives that can automatically take care of these mechanisms in a distributed control fashion using local handshake protocols is highly useful. Furthermore, this control is introduced by applying provable correct-by-construction transformations, and hence functionality is preserved on the speculative design. Thus, an automatable and scalable scheme for speculation is provided.

4.2 Speculative Execution

As it has been explained in the previous section, speculation is introduced by applying the following steps:

1. Find critical cycles from the output of early-evaluation multiplexors to their select input. If such cycle exists, speculation is the transformation of choice for increasing the performance.

Cycle	0	1	2	3	4	5	6
F_{in_0}	A	-	C	-	E	F	F
F_{out_0}	A	-	C	-	E	*	F
F_{in_1}	-	B	D	D	-	G	-
F_{out_1}	-	B	*	D	-	G	-
Sel	0	1	1	1	0	0	0
Sched	0	1	0	1	0	1	0
EB_{in}	A	B	*	D	E	*	G

Table 4.1: Example trace from Fig. 4.1(d). '*' = bubble in the channel, '-' = anti-token in the channel, otherwise the channel transmits a token.

This is because other transformations such as EB retiming, bubble insertion or early evaluation are useless, and multiplexor retiming alone typically introduces large area overhead.

2. Apply multiplexor retiming to move one or more logic blocks backward, out of the critical cycle.
3. Apply early evaluation to the retimed multiplexor. This transformation modifies the elastic controller, while the datapath stays the same.
4. Share the duplicated logic, introducing the speculation control that instantiates some prediction logic.
5. Apply retiming or bubble insertion to optimize the effective cycle time on the new configuration.

Table 4.1 shows a sample trace of the system from Fig. 4.1(d). F_{in_0} and F_{out_0} denote the input and output channel of the shared module F that serve the first input of the multiplexor, while F_{in_1} and F_{out_1} correspond to the second channel of the multiplexor. *Sel* is the select input of the multiplexor connected to the output of G functional block. *Sched* is the scheduling signal that carries the channel prediction for the shared unit F. Finally, EB_{in} is the data value at the input channel of the EB connected to the output of the multiplexor. In cycles $0, 1, 3, 4$, and 6, correct predictions are made (*Sel* = *Sched*). During these cycles, the early-evaluation multiplexor propagates the correct value, and an anti-token cancels the token waiting on the unused input channel. In cycles 2 and 5, however, mispredictions are made (*Sel* ≠ *Sched*). On mispredictions, the multiplexor stalls waiting for the required data. In cycle 2, channel 1 has been chosen by *Sel*, but F_{out_1} is not valid, since the selected token was not selected and it is being stalled at F_{in_1}. The stop bit on channel 1 is set by the

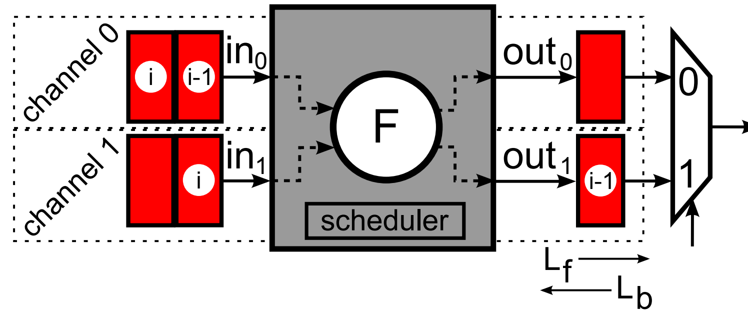


Figure 4.2: Example where the outcome of the early-evaluation multiplexor is predicted using the proposed method. Each one of the tokens is labeled with a counter (i or $i-1$). The stage of EBs after the shared unit has a forward latency of L_f and a backward latency of L_b.

multiplexor, and this way the shared unit can realize a misprediction has been made. Then, *Sched* is corrected during clock cycle 3.

The performance gain obtained by applying speculation is based on the assumption that the prediction can be done with a high probability of success. A good scheduler for a shared module is the one that can foretell which channel from the available ones must be used. It is the designers job to know how to take advantage of the circuit predictability. While the previous steps to introduce speculation are structural and can be easily automated, the design of a good scheduler may not trivial or even not possible in some cases.

In order to illustrate how speculative execution using sharing units works, consider the example in Fig. 4.2. There are two data channels sharing a functional module [1], and for simplicity of explanation, there is one EB at each input and one EB at each output of the shared module. Let us assume that the output EBs have forward latency L_f and backward latency L_b. A special case shown in the example in Fig. 4.1(d) (no buffers inserted) corresponds to the case of $L_f = 0$ and $L_b = 0$. Let us trace the processing of the i-th token, T_i, arriving at the input channel 0 of the shared module. The processing goes through 3 steps:

Propagating to the input of the shared module. Since token order is always preserved in elastic systems, for the i-th token, T_i, to be available at the input in_0 of the shared module, the $(i-1)$-th token in this channel, T_{i-1}, must have been processed or canceled by an anti-token. Let us assume that the speculation controller of the shared module predicted channel 1 and hence did not predict channel 0 during the previous transfer. Then, T_{i-1} is stalled at in_0, because it may be used in case of misprediction. If prediction of channel 1 was correct, an anti-token

[1] This example can be easily generalized for sharing of k blocks

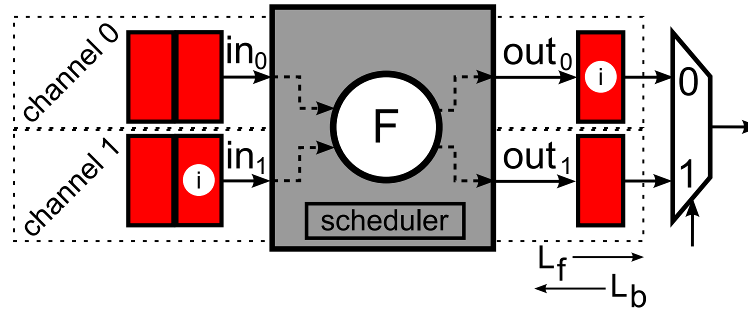


Figure 4.3: Example from Fig. 4.2 once token i in channel 0 has been propagated to the input of the multiplexor.

is generated by the controller of the multiplexor into channel 0. This anti-token propagates backwards reaching in_0 in L_b cycles and cancels T_{i-1}. Thus, the backward latency of EBs can affect the overall system performance and become a bottleneck. If the prediction was wrong, then the scheduler will have to let token T_{i-1} use the sharing unit since it holds the data required by the multiplexor for the $(i-1)$-th data transfer.

Prediction by the scheduler of the shared module. Once token T_i arrives to the input of the shared module, the scheduler may predict its channel and then T_i will get propagated through the shared module. Otherwise, the token T_i on channel 0 will be stalled until either the scheduler changes its prediction during one of the future cycles or an anti-token generated by the multiplexor arrives and cancels it out.

Early evaluation in the multiplexor. After token T_i is selected by the scheduler, it is transmitted through the shared module and then, stored by the output EB. After L_f cycles, the token reaches the input of the multiplexor, as shown in Fig. 4.3. If T_i was predicted correctly, once the select signal of the multiplexor becomes available, the early-evaluation multiplexor will generate a new token at its output. Otherwise, the token will be stalled at the input of the multiplexor, waiting for the correct token to arrive in the other channel.

4.3 EBs with Zero Backward Latency

As shown in the previous example, L_f determines the latency needed by tokens to reach the input of the multiplexor. In the multiplexor, the actual decision that was predicted takes place. Then, on

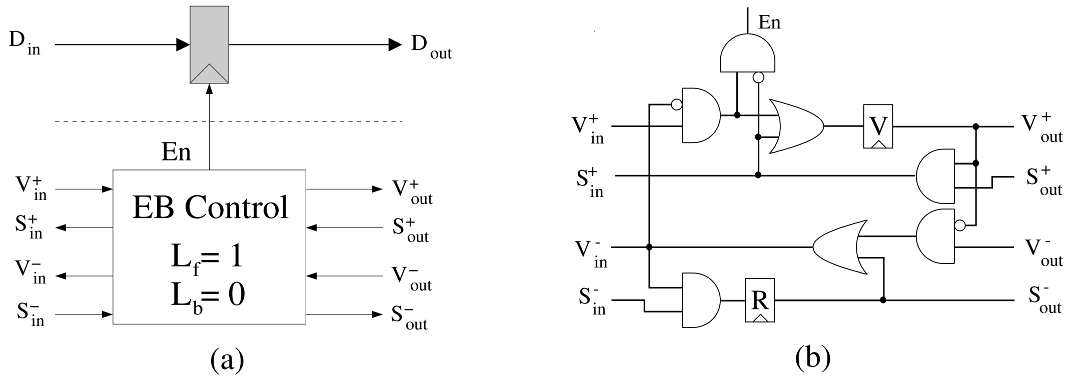


Figure 4.4: (a) Datapath and (b) control for an elastic buffer with no backward latency.

correct predictions, anti-tokens are sent backwards to nullify the unneeded tokens waiting in the input of the shared unit. This second latency is determined by L_b.

Anti-tokens must rush to the corresponding input channel of the shared module. Since the prediction was correct, the token at the input channel is no longer necessary. Each clock cycle elapsed since the multiplexor sends the anti-token until it reaches its final destination, the token waiting at the input channel is causing a traffic jam, forcing next tokens to be stalled, potentially increasing the queue size needed to store them and reducing the throughput.

Thus, it is possible to significantly improve the throughput of the elastic system by using EBs with $L_b = 0$ on the pipeline stages between the shared unit and the multiplexor making the decisions.

Figure 4.4(b) shows the control implementation for an EB with $L_b = 0$ and $L_f = 1$. The objective of this controller is to propagate the V^- bit as fast as possible. In order to maintain the integrity of the protocol, the stop bit S^+ must also be sent combinationally. The S^- and V^+ bits are stored in a flip-flop and keep a forward latency of one clock cycle.

This controller is much simpler than the regular EB controller with $L_b = 1$ and $L_f = 1$. However, its capacity is also smaller, it can only store one token ($C = L_f + L_b = 1$). Since there is no need to store two data items in the same EB at the same time, the datapath can be simplified and implemented with a single flip-flop, as shown in Fig. 4.4(a), instead of using a pair of latches with independent enable signals.

It can be verified that the controller in Fig. 4.4(b) complies with the SELF protocol and it is a refinement of the dual elastic buffer model discussed in Section 2.2.2. This implementation of EB can be used to reduce the throughput overhead of speculation and in other paths where fast

backward propagation is beneficial. However, a care must be taken not to chain too many of such controllers to avoid potentially long combinational delays in the control, or even combinational loops if they are connected forming a cycle.

4.4 Design Examples

So far, it has been assumed that speculation is used to improve the performance in a system which is tightly constrained by a cycle going through the select signal of a multiplexor. In this case, some modules can be removed from this cycle using Shannon decomposition and sharing. Then, the critical cycle through the multiplexor becomes smaller and easier to optimize, thus improving the performance of the system as long as it is possible to predict the behavior of the early-evaluation multiplexor.

In this section, the use of speculation combined with elastic systems is demonstrated on two interesting examples. In these examples, there is no critical cycle going through an early-evaluation multiplexor. However, speculation and early evaluation can still be used to improve their performance. First, a design with error detection and correction is presented. Speculation is used to start the computation before the correctness of the input data is known. On the second example, speculation is used to propose an efficient implementation for variable-latency units.

For the experiments the actual datapath of the examples was designed in Verilog, connected to the generated elastic controllers, and synthesized using commercial tools with a 65nm technology library. The final designs were timed using STA and simulated in order to obtain the actual effective cycle time.

4.4.1 Resilient Designs

Speculation can be used to add soft-error detection and correction in a pipeline without changing the performance of the system in case of error-free behavior. The single error correction and double error detection mechanism (SECDED) [153] is used as an example. For each 64 bits of data, 8 extra bits allow to detect and correct any single bit error. Besides, double bit errors are detected as well. Implementation details of SECDED can be found in [158].

Figure 4.5(a) shows an example with an adder where soft-error checking is done on each of its inputs [2]. Error correction needs a whole pipeline stage, and thus, the pipeline is deeper compared

[2]To simplify the figure, only one of the two inputs of the adder is drawn.

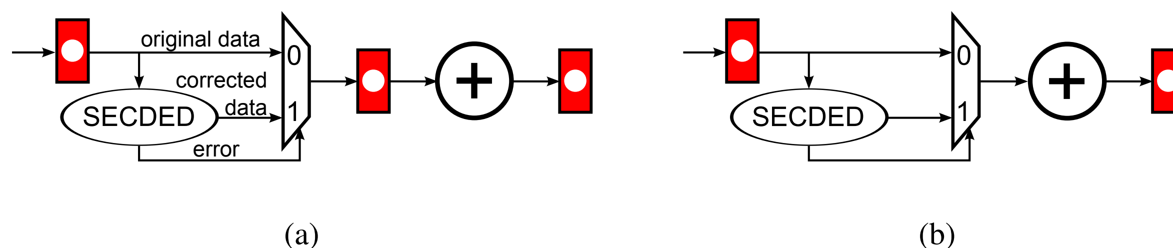


Figure 4.5: Speculation used for error correction and detection. (a) Original non-speculative resilient system, (b) design after removing one pipeline stage.

to a design with no error checking. By adding speculation to the design, the computation can start without waiting for SECDED to finish. If errors are detected, then the corrected value produced by the error correction module can be used, spending an extra cycle. Thus, speculation enables to achieve a shallower pipeline. This pipeline will probably have fewer data hazards, and hence it will improve performance.

Since we know that the speculative design will not need a pipeline stage dedicated to error correction, we can start by dropping this stage, as shown in Fig. 4.5(b). The EB that has been removed could have been retimed somewhere else or not even inserted initially. The cycle time in Fig. 4.5(b) is larger than the one in Fig. 4.5(a), due to the path going through the error correction unit plus the adder.

Next, Shannon decomposition can be applied to the adder to move it to the other side of the multiplexor. Since the error signal from the error detection unit going to the early-evaluation multiplexor is critical in timing, it is better to retime the output EB and move the multiplexor to the next pipeline stage. If possible, the select signal of early-evaluation multiplexors should not have a tight timing. Then, a bubble is added between the error correction unit and the adder, to break this long combinational path.transformed into a shared unit. The final design is shown in Fig. 4.6.

Since errors are expected to be unusual, the scheduler of the shared unit always predicts that the input data will be correct and the execution of the addition starts normally. At the same time, SECDED is computed on both inputs to detect errors in the input data. The bubble at the data output of the error unit allows to store the corrected data in case of error. Next cycle, if SECDED detected an error, the mispredicted computation is stalled at the input of the multiplexor, and the addition is restarted using the corrected values coming from the SECDED unit. Thus, the scheduler must only listen to the outcome of the SECDED unit to decide which decision to make. If there

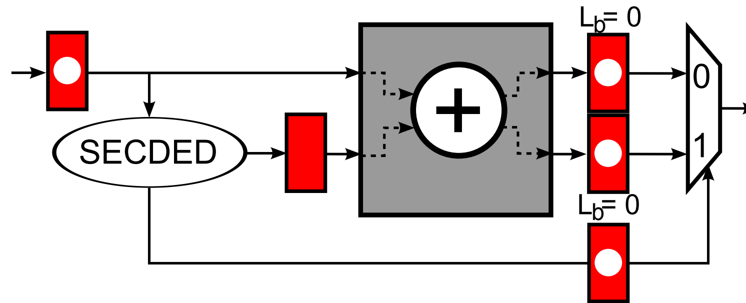


Figure 4.6: Final speculative design corresponding to the original system in Fig. 4.5(a).

were errors on the last cycle, the addition is replayed with corrected values, otherwise, a new operation is started.

This design has been synthesized using a 64-bit prefix-adder, and it has been checked that there is no performance penalty during the error-free behaviors. Whenever an error is detected, a single clock cycle is lost in order to repeat the computation with the corrected data. The EBs between the shared unit and the early-evaluation multiplexor must have zero backward latency. Otherwise the penalty to correct errors would be two cycles.

This mechanism can also be used for error-protection of memories and register files. The area overhead due to speculation in Fig. 4.6 is 36%, caused mainly by the recovery EBs necessary for speculation. Notice that this overhead is paid on a single pipeline stage, and hence, it would be amortized across the whole system when implemented on a real pipeline.

4.4.2 Variable-Latency Unit

Variable-latency units, such as telescopic units [12], optimize the frequent paths of the design into a faster single clock cycle, and execute infrequent critical paths in two or more clock cycles.

Variable latency in elastic systems can be handled in a natural way thanks to their handshake protocol. Figure 4.7(a) shows how a generic variable-latency unit with a 3 signal protocol can be connected to an elastic system. When a valid token arrives, the *Go* signal is asserted to the variable-latency unit. Once the job is ready, the variable-latency unit will assert the *Done* signal, which will trigger a token transmission to the output. The variable-latency unit must hold the output value until the *Ack* signal is asserted, and then it can continue to the next computation. *Ack* is asserted if there is a token transfer or if anti-token goes through the variable-latency unit, canceling the on-going computation.

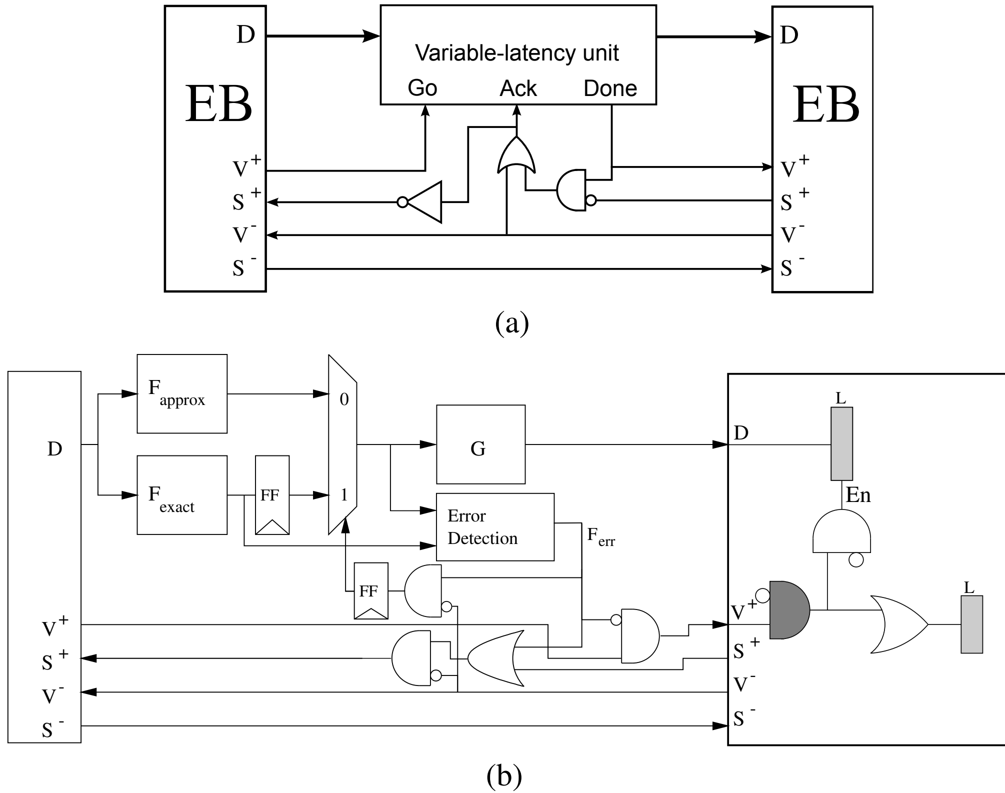


Figure 4.7: (a) Protocol to synchronize variable-latency units with elastic systems, (b) example of a generic variable-latency unit which takes 1 or 2 clock cycles.

Figure 4.7(b) shows the datapath and control details for a variable-latency unit which can take 1 or 2 clock cycles to compute. F_{approx} is an approximation of F_{exact} that can be obtained automatically [9], and it has a shorter critical path. They can both be connected to the second part of the computation, called G in this example. Most of the computation cycles, the approximation is correct ($F_{approx} = F_{exact}$), and thus, $F_{err} = 0$. Therefore, the function can be computed within a single clock cycle, and the controller does not stall the token flow. However, when the approximation is incorrect, F_{err} inserts a bubble into the receiver channel and stalls the sender. The next cycle F_{exact} can be used to finish the computation. The communication between the datapath and the elastic controller is equivalent to the one in Fig. 4.7(a).

In Fig. 4.7(b), F_{err} is connected directly to the controller, which handles the clock gating mechanism to govern the datapath. The path from the V^+ signal to the active low latches inside the elastic buffer are shown in the figure. It is possible that F_{exact} followed by the comparison to F_{approx} plus the control gates until the latches become critical, specially if the datapath width is

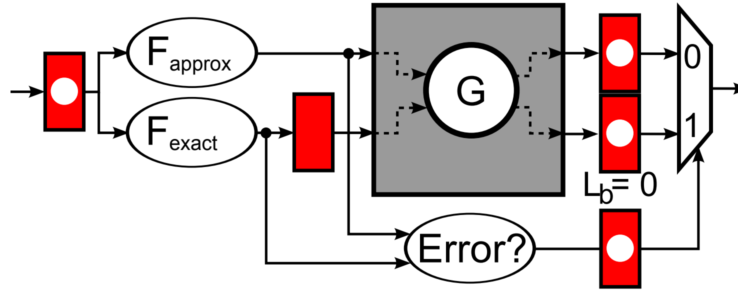


Figure 4.8: Speculation used for variable latency.

large. Critical paths involving both datapath and control can be difficult to handle.

An alternative implementation that does not have this problem can be based on using speculation with replay in case of an error. The system in Fig. 4.8 always speculates that the approximate computation is always correct (in which case the computation is finished in one clock cycle). G is shared between the channel coming from F_{approx} and the channel coming from F_{exact}. On fast executions, the early-evaluation multiplexor selects the path coming from F_{approx}, and a fast anti-token is sent through the EB with $L_b = 0$ in order to cancel the token from F_{exact}, which will be stored in the bubble. Whenever F_{err} is asserted, the prediction was incorrect. Then, the next clock cycle, the speculation controller uses the result of F_{exact} computation stored in the bubble, while the early-evaluation multiplexor stalls waiting for the correct data. With this approach, the critical path is taken out of the elastic controller.

A variable-latency ALU has been implemented using a simple pipeline with an 8-bit datapath. In the initial system, shown in Fig. 4.7(b), F_{err} is part of the critical path, going into the latches of the control. This problem is gone in the final design shown in Fig. 4.8. The speculative design improves the effective cycle time by 9% with a 12% area overhead. The area overhead is due to extra EBs storing the results after the shared unit. Furthermore, physical synthesis becomes simpler because the critical path does not end at a clock gating cell inside the controller.

4.5 Conclusion

This chapter has proposed a novel method for applying speculation in elastic systems, leveraging the benefits of early evaluation and token counterflow. It is performed by applying Shannon decomposition and module sharing to a non-speculative design. Since both transformations are correct-by-construction, functional equivalence is preserved when applying speculation.

This method can provide a performance improvement on designs where the critical path lies in a cycle that goes from the output of an early-evaluation multiplexor to its control signal. If this cycle is both critical in cycle time and in throughput, it may not be possible to improve the performance by adding bubbles to further pipeline it. The proposed method allows to start execution of a part of the cycle speculatively, before it is known what of the input data will be selected by the early-evaluation multiplexor.

In order to do so, some logic blocks from the critical cycle are retimed out of the critical cycle to the input data of the multiplexor. These blocks are then shared by the elastic channels that provide data inputs to the critical early-evaluation multiplexor. A local scheduler decides which of the channels should be allowed to use the shared unit, trying to predict the selection of the early-evaluation multiplexor. The performance speed-up greatly depends on the quality of the scheduler.

It has been shown that speculation can be used to enhance performance of two realistic examples. Speculation can be used to optimize variable-latency units, removing critical signals out of the control logic. Furthermore, it can be used to implement resilient designs that start performing useful computation before the outcome of the error-checking unit is known, thus achieving a higher throughput.

Chapter 5

Automatic Pipelining

In pipelined circuits, the structure and the boundaries of the pipeline is one of the most important decisions made early on the design process. This choice will have a direct impact on the performance and complexity of the final design. Furthermore, a poorly made decision may force a redesign later in the design process, with the corresponding increase in the cost of the circuit and penalty in time to market. However, pipeline design is often done ad hoc, due to the significant computational costs of simulation during the design space exploration and the lack of analytical optimization methods capable of pipelining in the presence of dependencies between iterations.

Elastic systems provide many of the features required in order to implement efficient pipelines. The distributed handshake protocol controls data flow and can locally stall communications when some data are not available. Early evaluation can be used to handle the control for the forwarding paths, needed to maintain execution parallelism even in presence of data dependencies. Furthermore, the set of elastic transformations presented in Chapter 3 provide a great set of tools for design optimization.

This chapter presents a method that automates pipeline exploration using these transformations. The presented method combines mathematical models, such as retiming and recycling with early evaluation [24], with other heuristic methods. While manual crafting of an interlocked pipeline can be laborious and error prone, this method guarantees a provably-correct construction because all the used transformations are correct-by-construction.

Design space exploration is driven by the expected workload for the circuit. Thus, the pipeline is optimized knowing which parts of the circuit will be executed more often and what is the expected frequency for events such as data dependencies or instruction branches. This information

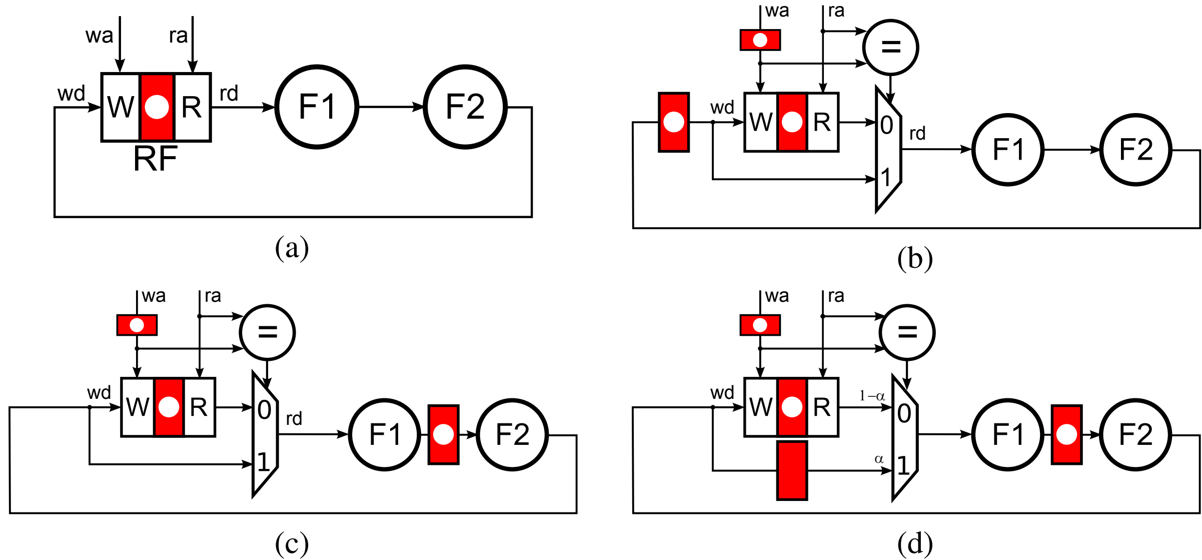


Figure 5.1: Simple pipelining example. (a) Initial functional specification, (b) design after applying bypass to the register file, (c) retiming of the elastic buffer, and (d) bubble insertion in the bypass path.

is very useful for optimization with elastic circuits, since it allows to adapt the latency of the computations and communications so that it is optimized to the expected workload.

The contributions presented in this chapter have been published in several conference and workshop papers [48, 65, 66, 87]. This chapter first discusses how an unpipelined functional description of a microarchitecture can be pipelined by applying a sequence of elastic transformations. Next, a method for automatic microarchitectural exploration is discussed and the efficacy of this approach is illustrated with some pipelining examples.

5.1 Introduction

Starting with a functional specification graph of a design, it is possible to pipeline it by using elastic transformations [87]. Bypasses with early-evaluation multiplexors are essential for pipelining, since they introduce new EBs that can be retimed backwards. In order to pipeline a design, bypasses must be inserted around register files and memories of the functional model. Then, the graph is modified to enable forwarding to the bypass multiplexors. 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. Speculation and insertion of variable-latency units can also be considered in

the exploration framework.

Let us show how on the example in Fig. 5.1(a), which computes $RF[wa] = F2(F1(RF[ra]))$ at each clock cycle. Ignoring the delay of register reads and writes, the cycle period of the basic architecture (Fig. 5.1(a)) is determined by the sum of delays of $F1$ and $F2$. Let us assume that the two delays are equal (D), then the cycle time is $2D$. The bypass transformation (explained in Section 3.4.1) incorporates new registers in the circuit and extra logic (multiplexor and comparison) to determine when the data must be read from the register file or from the bypass. By retiming the new register, the circuit can be transformed from Fig. 5.1(b) to Fig. 5.1(c). Still, the period is determined by a critical path that starts at the input of $F2$ and ends at the output of $F1$ through the bypass. Assuming that the multiplexor delay is negligible, the cycle time is still $2D$.

Here is where *elasticity* comes into play. A bubble can be inserted at one of the inputs of the multiplexor, as shown in Fig. 5.1(d). Now the cycle time is D. However, the bubble also has a negative impact on the performance. If early evaluation is not used, the throughput is determined by the cycle containing two registers and one token. This results in processing only one data item every $2D$ time units. Hence the effective performance does not improve.

By using early evaluation, elastic pipelines attain optimal average-case performance by dynamically selecting bypasses using early enabling multiplexors. When no data hazards are present, the multiplexor propagates the data read from the register file and an anti-token is created in the bypass path to kill the forwarded data, as shown in Fig. 5.2(a). However, when a bypass must be selected, the multiplexor cannot be enabled, and the system stalls waiting for the bypass data, as shown in Fig. 5.2(b).

The exact throughput of the system (computed using Markov chains) is $1/(1 + \alpha)$, where α is the bypass probability (data hazard). Therefore, the speed-up of the pipelined microarchitecture is $2/(1 + \alpha)$. Figure 5.3 shows the relationship between the bypass probability and the speed-up in this example.

5.2 Correct-by-construction Pipelining of a Simple Example

More complex transformations are often required in order to derive deeper pipelines. Figure 5.4(a) shows an example with two independent functions (M and ALU), each one with a different depth. The register file RF is the only state holding block. IFD fetches instructions and decodes the opcode and register addresses. The results are selected by the multiplexor for RF write-back. M has been divided into three nodes. The breaking up of logic to allow pipelining is a design decision

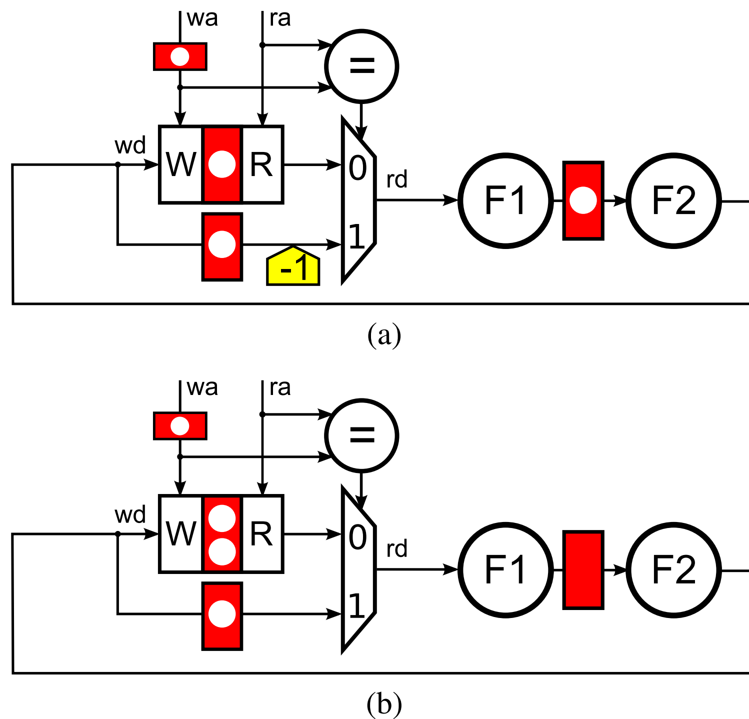


Figure 5.2: Simulation by case, (a) when bypass path is not needed, (b) when the register file is stalled waiting for a token in the bypass path.

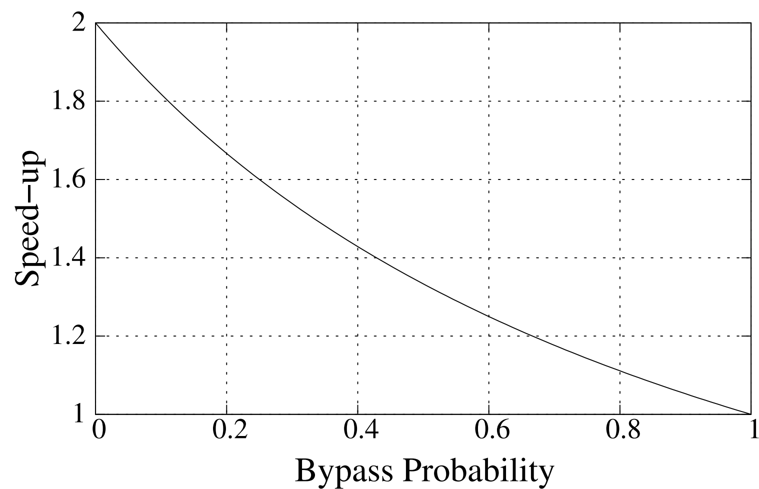


Figure 5.3: Speed-up achieved by pipelining the example in Fig. 5.1(d) in function of the bypass probability.

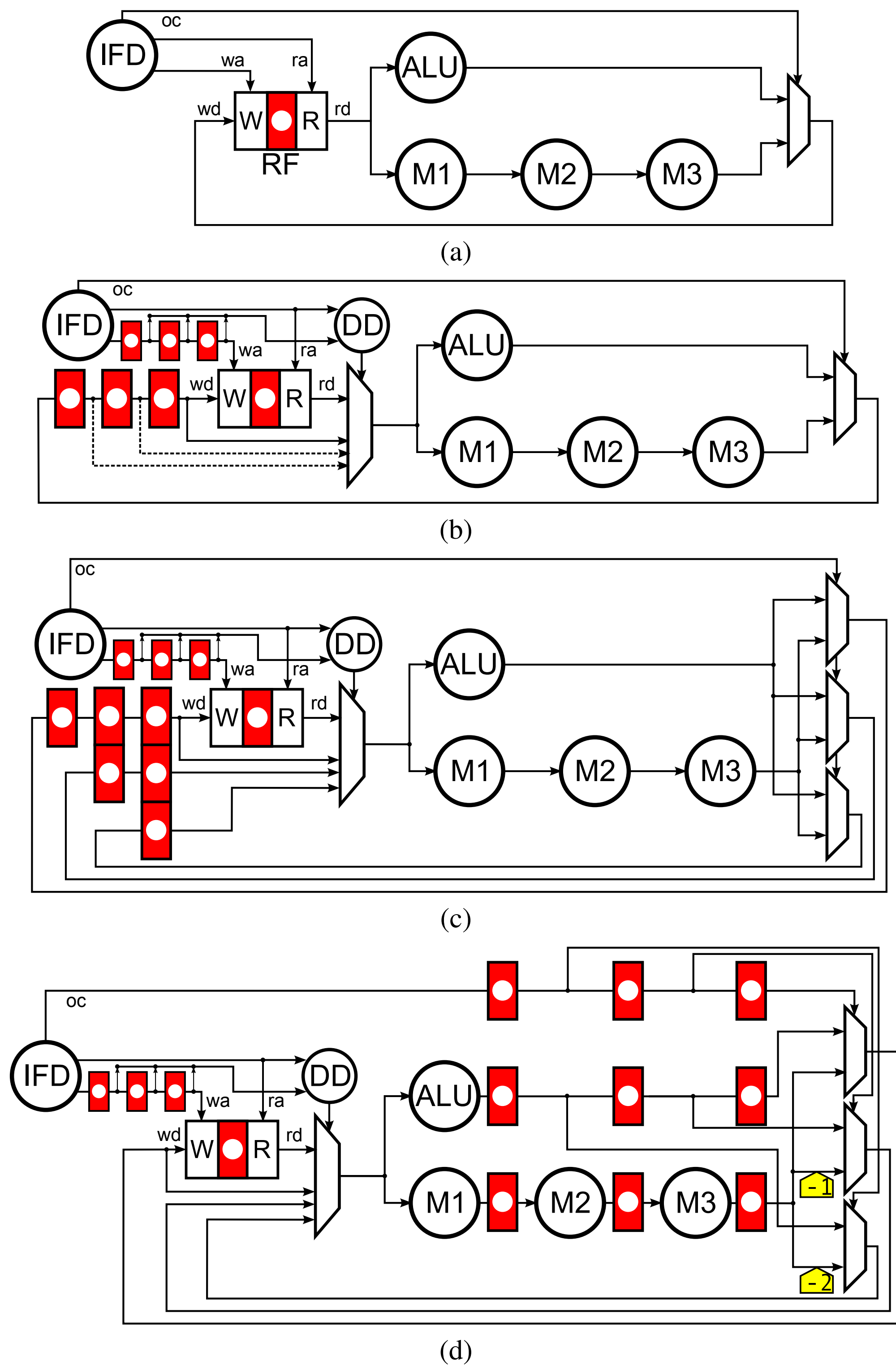


Figure 5.4: (a) Graph model of a simple design with two instructions, (b) After 3 bypasses, (c) Duplicate mux, enable forwarding, (d) Final pipeline after transformations.

that is typically considered in concert with pipelining decisions. Thus, the user may try to divide a functional block into several nodes and let the optimization algorithm decide the best channels to place the EBs.

In Fig. 5.4(b), the bypass transform has been applied three times on *RF* to build 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 multiplexor must be selected. The conventional use of bypasses is to forward data already computed to the read port of the bypassed memory element. In addition, this bypass network can be used as a data hazard controller, taking advantage of the underlying elastic handshake protocol with early evaluation to handle stalls.

Notice that the EB closer to the register file cannot be retimed backwards in order to pipeline the design. One way to enable retiming of this EB would be to apply anti-token insertion from Fig. 3.11 on the dashed forwarding paths. However, adding an anti-token on the dashed path would stall the system every time there is a back to back data dependency, leading to a throughput degradation.

A better solution is to duplicate the *WB* multiplexor (the right-most one) and duplicate the bypass EBs so that forwarding paths from the output of the functional units to the bypass multiplexor are created, as shown in Fig. 5.4(c) [1]. Finally, by applying retiming and anti-token insertion transformations on Fig. 5.4(c), the pipeline in Fig. 5.4(d) can be achieved. 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, like *ALU* 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 multiplexors. This situation corresponds to a read after write (RAW) dependency involving a result computed by *M*, which needs three cycles to complete.

5.3 Exploration Engine

The previous section shows how pipelining can be achieved by applying transformations one by one. For a given elastic system, there are many different possible pipelines. For example, Fig. 5.5

[1] See Section 5.3.2 for an example which shows why forwarding paths are needed

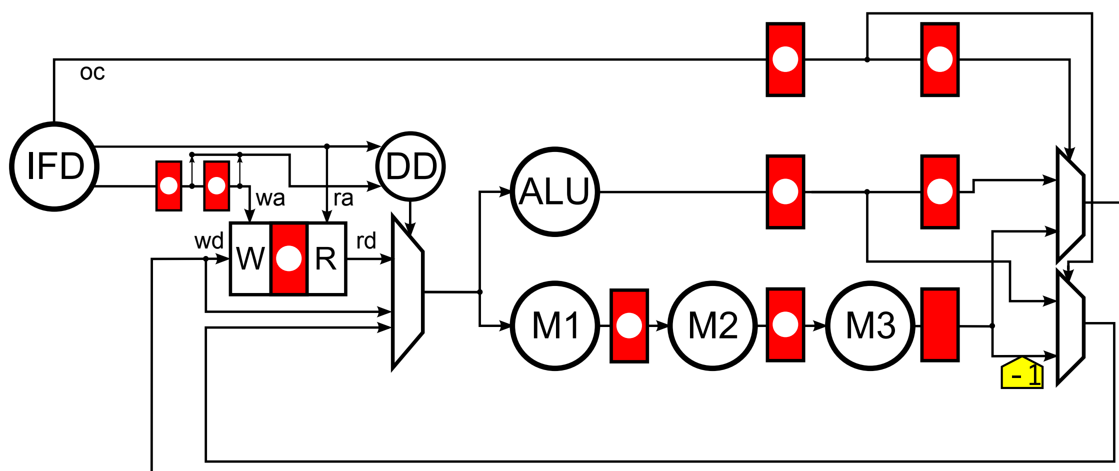


Figure 5.5: Another possible pipeline for the graph in Fig. 5.4(a).

shows an alternative pipeline for the design in Fig. 5.4(a). In this example, only two bypasses have been applied to the register file. Then, in order to fully pipeline *M* operation, a bubble is used in the final stage.

At first sight, it is difficult for a human to determine whether Fig. 5.5 is better than Fig. 5.4(d). Figure 5.5 provides the same cycle time or slightly better (since there are fewer bypasses and hence fewer multiplexors) than Fig. 5.4(d) and a smaller area (there are fewer `EB`s). However, the cycles going through *M* have a worse token/delay ratio in Fig. 5.5, and hence a worse throughput. The effective cycle time of Fig. 5.5 will be better when *M* is rarely used, but it will be worse if it is used too often.

A designer performing manual exploration may miss some of these configurations with better performance or a more interesting performance/area trade-off. Furthermore, manual exploration is tedious and it becomes harder as the microarchitectural graph grows. Pipelining a system with hundreds of nodes manually is close to impossible.

An algorithm that automatically explores the design space can potentially find better designs, or at least, provide better confidence on the value of the solution. This section proposes an automatic exploration framework. First, the retiming and recycling optimization is presented. Retiming and recycling is used in the inner loop of the proposed method. Then, it is shown how to add bypasses and forwarding paths to microarchitectural graphs. Finally, the exploration methodology is discussed and an exploration algorithm is proposed.

5.3.1 Retiming and Recycling

This section reviews the retiming and recycling (RR) problem [24], which is used by the inner loop of the method presented in this chapter. RR formalizes a subset of the elastic transformations presented in Chapter 3 (early evaluation, retiming of EBs, insertion of empty EBs, insertion and retiming of anti-tokens) and solves an optimization problem on this formalization in order to find the configuration with the best effective cycle time. This problem was initially proposed in [35] and solved as a mixed integer linear programming problem (MILP) in [26]. Finally, it was extended to support early evaluation and anti-tokens in [24].

Given a microarchitectural graph $G(N, E)$ and an initial number of tokens on each edge defined by a vector $R_0 \in \mathbb{Z}^{|E|}$, a *retiming and recycling configuration*, RC, is a pair of vectors $R'_0 \in \mathbb{Z}^{|E|}$, $R' \in \mathbb{Z}^{+|E|}$, where R'_0 assigns a number of tokens to each edge and R' assigns a number of EBs to each edge. For each edge e, there must be enough EBs to store all tokens in the edge, i.e., $R'(e) \geq R'_0(e)$. R_0 and R'_0 can be negative, which represents anti-tokens. The capacity of all EBs is assumed to be large enough. Furthermore, each node n is assigned a combinational delay $\delta(n)$ and a latency $lat(n)$.

A *retiming vector* $r \in \mathbb{Z}^{|N|}$, is a map $N \to \mathbb{Z}$ that for every edge $e = (u, v)$ transforms R_0 to R'_0 as follows: $R'_0(e) = R_0(e) + r(v) - r(u)$. Thus, $r(u)$ indicates how many tokens (or anti-tokens) are retimed through node u and in which direction.

Some elastic transformations presented in Chapter 3 can be represented on an RC using a retiming vector r. Bubble insertion on edge e can be achieved by incrementing $R'(e)$. Given two edges $e_0 = (u, v), e_1 = (v, z)$ such that $R_0(e_1) = 0$, anti-token insertion is represented by $r(v) = 1$: this adds one token to e_0 and inserts one anti-token to e_1. This new token inserted on e_0 with anti-token insertion transform can be then retimed through node u. Backward retiming of tokens is achieved by incrementing $r(v)$, and forward retiming by decrementing $r(v)$. Analogously, backward retiming of anti-tokens is achieved by decrementing $r(v)$ and forward retiming of anti-tokens is achieved by incrementing $r(u)$. Thus, insertion, retiming and grouping of anti-tokens can also be easily modeled using retiming vectors.

Finally, the cycle time of the system, τ, can be constrained using a set of linear equations called `Path_Constr`(RC, τ), and the inverse of the throughput, $x = 1/\Theta$, can also be constrained using a set of linear equations called `Thr_Constr`(RC, x). These sets of equations are defined in [23, 24]. Finally, the function to optimize is the effective cycle time $\xi = \tau/\Theta = \tau \times x$. The final problem to

solve is :

$$\begin{aligned} & Minimize \quad x \cdot \tau, \\ & R_0'(e) = R_0(e) + r(v) - r(u), \\ & R' \geq R_0, R' \geq 0, \\ & \texttt{Path_Constr}(\mathrm{RC}, \tau), \\ & \texttt{Thr_Constr}(\mathrm{RC}, x), \end{aligned} \tag{5.1}$$

Since $\texttt{Thr_Constr}(\mathrm{RC}, x)$ is an upper bound approximation of the throughput, the exact solution of Equation (5.1) is not necessarily the one with the minimum effective cycle time, but it is typically a very good approximation. On the other hand, Equation (5.1) represents a big challenge for existing solvers, since the objective function is not convex. A faster heuristic method based on mixed integer linear programming is presented in [24]. Given two RCs, RC_1 and RC_2, RC_1 dominates RC_2 if $\Theta(\mathrm{RC}_1) > \Theta(\mathrm{RC}_2)$ and $\tau(\mathrm{RC}_1) \leq \tau(\mathrm{RC}_2)$.

The heuristic method to solve Equation 5.1 finds the set of *non-dominated* RCs. Each RC can be considered a Pareto point which explores a different cycle time / throughput trade-off[2]. The first Pareto point found by the heuristic has the minimum possible cycle time, equal to the maximum node delay. Then, each Pareto point has a larger cycle time, and also a larger throughput. The last point corresponds to the min-delay retiming configuration, which has no bubbles and always has throughput equal to 1.

For example, Fig. 5.6 shows an example that has three non-dominated Pareto-point configurations. The first Pareto point is the optimal one, since its effective cycle time is 2, compared to 3 for the other two configurations. If the early-evaluation multiplexor had different probabilities at the inputs, the configurations may be different, or they may be the same but the optimal one in terms of effective cycle time could be a different one.

5.3.2 Bypassing and Forwarding

Figure 5.7(a) shows a register file with 3 bypasses. Write address and read address are omitted for simplicity. Even though three EBs have been inserted, only the leftmost one can be retimed backwards. Bypasses alone are not sufficient to pipeline cyclic systems.

By using the anti-token insertion multiple times, EBs and anti-tokens can be inserted on the bypass channel. Then, all the EBs can be retimed out of the bypass structure, as shown in Fig. 5.7(b), and used to pipeline the design along the dotted line. However, the inserted anti-tokens will stall

[2]Strictly speaking Pareto points are formed by the pairs $(\Theta(\mathrm{RC}), -\tau(\mathrm{RC}))$

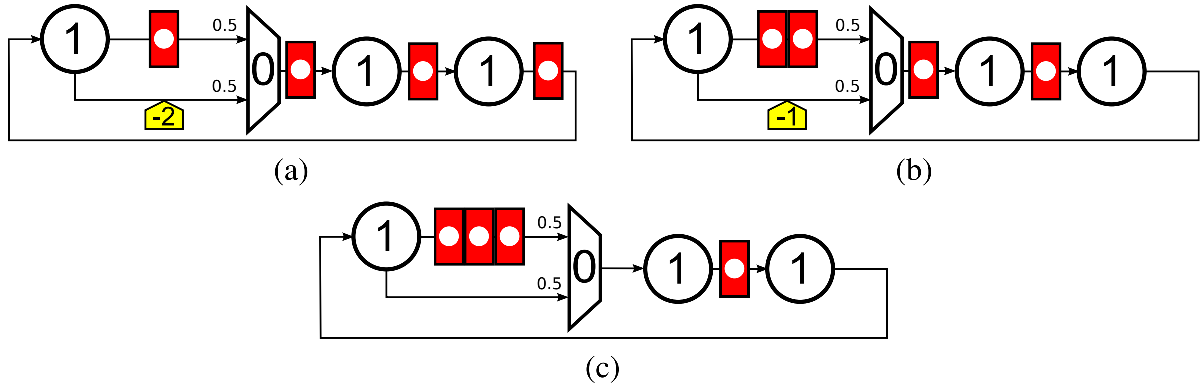


Figure 5.6: Example with three non-dominated Pareto-point configurations found by retiming and recycling. Their performance metrics are (a) $\tau = 1, \Theta = 0.5, \xi = 2$, (b) $\tau = 2, \Theta = 0.66, \xi = 3$, (c) $\tau = 3, \Theta = 1, \xi = 3$.

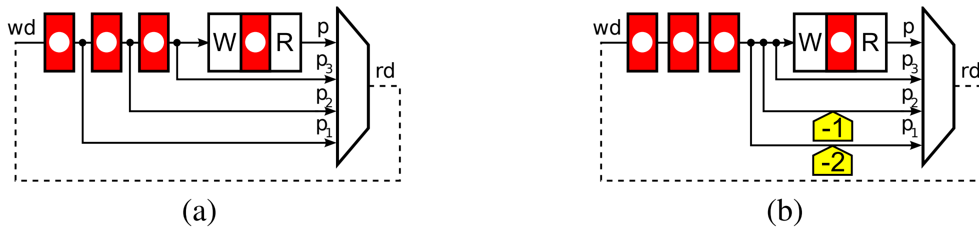


Figure 5.7: 3 bypasses (a) before retiming, (b) after retiming using anti-tokens.

the system on data hazards, waiting for the correct token to arrive.

Data hazards can also be solved by forwarding the required value if it is already available somewhere within the pipeline. In order to enable forwarding to the bypass multiplexor, some EBs and multiplexors must be duplicated to create new paths.

For example, after bypassing the register file three times in the graph in Fig. 5.4(b), some nodes in the graph are duplicated to achieve the system in Fig. 5.4(c). In this figure, each bypass is fed in-

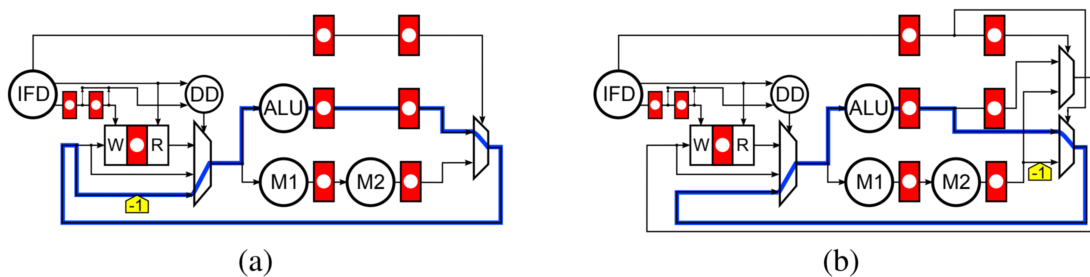


Figure 5.8: Example pipeline (a) without using forwarding paths, (b) using forwarding paths.

dependently, creating new forwarding paths and retiming opportunities that can lead to the retimed design in Fig. 5.4(d). In order to automatically create all the possible forwarding paths so that they can be used by RR optimization, it is necessary to traverse the microarchitectural graph backwards starting from the register file, and duplicate all the fork nodes, with the corresponding buffers, until the write-back multiplexor is found. The write-back multiplexor should also be duplicated. These fork nodes are drawn as small points in Fig. 5.7(a).

Figure 5.8(a) shows an example similar to the one in Fig. 5.4, but the M execution unit has only 2 stages instead of 1 (to make it simpler). In this example, no forwarding paths have been added. The pipelining has been built just by applying anti-token insertion and retiming. Figure 5.8(b) shows the same example where the forwarding paths have been added. In both figures, the cycle that corresponds to a back-to-back dependency between consecutive ALU instructions has been highlighted. In the example with forwarding paths, this cycle has a throughput of 1. However, in the example without forwarding paths, this cycle has a throughput of 0.5, since it has 1 token (2 tokens and 1 anti-token) and two EBs. Therefore, the example with forwarding paths will have a higher throughput as long as this cycle does not have 0 probability.

Furthermore, it is not possible to achieve a throughput of 1 for this cycle without adding the corresponding forwarding path. It is the only way to provide the data produced by the ALU as soon as possible. Using the same write-back multiplexor that provides data to the register file means delaying the data transmission, with the corresponding performance penalty. After running the retiming and recycling optimization, some of the fork nodes and early-evaluation multiplexors created in order to enable some forwarding paths may be merged back if they have not been fully used (e.g., the three write-back multiplexors in Fig. 5.4(c) could be merged back since they have the same inputs with the same exact latency).

In order to be able to analyze the performance of a microarchitectural graph without simulating, it is necessary to assign probabilities to the inputs of early-evaluation multiplexors. For bypass structures, these probabilities should be determined by the expected frequencies of data hazards in the class of workloads for which optimization is done.

In this work, data hazard probability is modeled on bypass multiplexors with a single probability, γ, which represents the probability that the register file reads the value written at the previous clock cycle (back-to-back dependency). Then, further dependencies are assumed to decrease geometrically. Thus, the probability of p_1 (from Fig. 5.7(a)) to be selected by the multiplexor is $p(p_1) = \gamma$; the probability that the register file reads the data value written two clock cycles before, i.e., distance two dependency, is $p(p_2) = (1 - \gamma)\gamma$; and in general, the probability of distance i

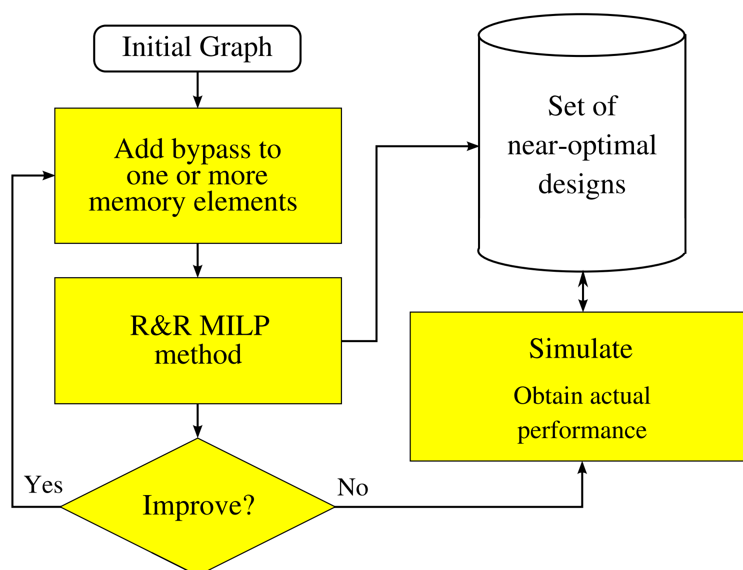


Figure 5.9: High level view of the exploration algorithm.

dependency is $p(p_i) = (1-\gamma)^{i-1}\gamma$. The path from the register file directly to the multiplexor (no dependencies) is taken when no other path is selected.

5.3.3 Two-phase Exploration

Simulations of Verilog controllers can take significant time: up to 30 minutes for a several hundreds node example and 10K simulation cycles. Thus, it is not feasible to simulate each of the RCs found by RR executions, because the exploration time would become too large.

Depending on the number of early-evaluation nodes and the number of cycles in a graph, each RR execution may find up to 7 or 8 different Pareto points. For example, if there are two elements that can be bypassed, and the algorithm tries up to 11 bypasses to each one, there can be close to a thousand of different configurations. In order to prune the design space, bypasses are applied greedily instead of trying all combinations, and the throughput is estimated rather than simulated during the first stage of the exploration.

Hence, design space is explored using a two-phase exploration strategy, similar to [62]. During the first phase, bypasses are applied incrementally on the memory elements and RR optimization executed on each step, as shown in Fig. 5.9. Within RR, performance for each point is estimated using analytical analysis, and the most promising points are stored. At the end of the exploration, each of the design points with near-optimal performance is simulated in order to determine their

throughput with better accuracy. Finally, the method returns the configuration with the best effective cycle time or a set of Pareto-point configurations with different effective cycle time and area.

Since the ordering determined by throughput analysis and by simulation are similar (see Section 6.5.2), it can be safely assumed that design points pruned during the first phase are not optimal.

5.3.4 Exploration Algorithm

During the exploration phase, Algorithm 1 tries different number of bypasses for each of the memory nodes of the graph. Even though the number of elements in a microarchitecture that can be bypassed is not typically large, it is interesting to be able to support an undetermined number of them. If there is a single register file, the algorithm adds one bypass at each iteration, adds the corresponding forwarding paths and then calls the retiming and recycling function (RR) for performance optimization.

The more bypasses the algorithm adds, the less recycling is needed for achieving a small cycle time through pipelining. Therefore, the throughput (and the effective cycle time) keeps improving at each iteration, while the cycle times can be kept constant. At some point, either the number of bypasses is enough to fully pipeline the design using retiming, or the throughput degradation due to data hazards stalls in the new bypass is larger than the cycle time improvement due to pipelining. At this point, adding more bypasses does not improve the performance, and the exploration is completed.

If the graph has several memory elements, the exploration can be performed in a similar way, but the algorithm selects which element to bypass next based on sensitivity analysis. Given a graph G with a set of elements that can be bypassed (*memories*(G)), Algorithm 1 greedily selects which element to bypass. For every memory m, *H.bypass*(m) applies one more bypass and adds the corresponding forwarding paths. Then, the algorithm calls RR optimization and estimates the effective cycle time ($\xi^{lp}(c)$) for each Pareto point c found by RR.

The node m that leads to the fastest configuration is chosen as the next step, and the greedy algorithm continues into the next iteration. Variable ξ_{loop} keeps track of the best effective cycle time and G_{loop} stores the best graph found so far within the loop of the algorithm.

When the estimated effective cycle time cannot be improved with a given threshold (called *improve_threshold* in the algorithm), the exploration stops. The priority queue *explored_points* stores the best designs within a given performance overhead compared to the best design found by analytical estimation. At the end, all stored designs are simulated in order to find the best one. In

Algorithm 1: Bypass_One(G).

```
explored_points := ∅        #Set to keep the most promising design points
ξ_min := ∞                  #Best effective cycle time found
done := false               #Boolean to indicate when exploration is over
while not done do
    ξ_loop := ξ_0 := ξ_min   #Best effective cycle time within the loop
    for m ∈ memories(G) do
        H := G
        H.bypass(m)          #Adds one bypass plus forwarding paths to memory m
        RCs := RR(H)         #Returns all the Pareto-point configurations found
        for c ∈ RCs do
            explored_points.add(c)      #Store the design point if it is promising
            ξ_min := min(ξ^lp(c), ξ_min)
            if ξ^lp(c) < ξ_loop then
                #New best configuration found within the loop
                G_loop := H
                ξ_loop := ξ^lp(c)
    done := ξ_min ≥ ξ_0 * (1 − improve_threshold)
    G := G_loop          #Recover best configuration for next iteration
return G, explored_points
```

addition to the best design point, with effective cycle time $\xi_{\min}$, the frontier of best solutions is kept, because the analytical LP formulation for the effective cycle analysis is approximate. Therefore, some other points in the frontier close to the best estimated may have the best performance (as can be checked by simulation). The ability to store a set of solutions can be also used to extend the algorithm for generating Pareto points in the (performance, area) solution space.

Some graphs may require several memory elements to be bypassed at the same time in order to reach a performance improvement. On such designs, the previous algorithm may be inefficient. Therefore, Algorithm 2, which bypasses all memory nodes at the same time before RR is called, is executed first. After a local minimum is found by Algorithm 2, Algorithm 1 is called for finer grain tuning.

If a memory node is not further bypassed by the Algorithm 1, it might have too many bypasses. For example, say a graph has two memory elements and the optimal configuration has 3 bypasses for the first memory and 9 for the second. Assume that Algorithm 2 bypasses the graph until (7,7), and then Algorithm 1 continues until (7,9). Extra bypasses do not degrade the performance of the system, since the same pipelining can be achieved for (7,9) and for (3,9). Configuration

Algorithm 2: Bypass_All(G).

```
explored_points := ∅
ξ_min := ∞
done := false
while not done do
    ξ_0 := ξ_min
    for m ∈ memories(G) do
        G.bypass(m)
    RCs := RR(G)
    for c ∈ RCs do
        explored_points.add(c)
        ξ_min := min(ξ^lp(c), ξ_min)
    done := ξ_min ≥ ξ_0 ∗ (1 − improve_threshold)
return G, explored_points
```

(7,9) contains unnecessary bypasses. By running Algorithm 1 again starting from the best design found, (7,9), but using *unbypass* transformation instead of bypass, the design space can be further explored. Algorithm 1 with unbypass has a different termination condition: instead of checking that a better design point is found, it checks that the best RR Pareto point is not significantly worse than the best ξ found so far.

Algorithm 3: Top-level Exploration Algorithm.

```
G_best, explored_points_1 := Bypass_All(G)
G_best, explored_points_2 := Bypass_One(G_best)
G_best := simulate(explored_points_1 + explored_points_2)
H, explored_points := Unbypass_One(G_best)
G_best_2 := simulate(explored_points)
```

The top-level algorithm, shown in Algorithm 3, calls Algorithm Bypass_All first and then Algorithm Bypass_One. Next, the best design points are simulated in order to obtain the best one from this first exploration. Since it may be over-bypassed, Algorithm Bypass_One with unbypass transformation is called, to further explore new design points. At the end, this new explored points are simulated. The performance of G_{best_2} can be at most equal to G_{best}. It has not been observed a large performance difference between them in any of the experiments. Depending on the area gain and the performance loss of G_{best_2}, the user can decide which one to take as the best one.

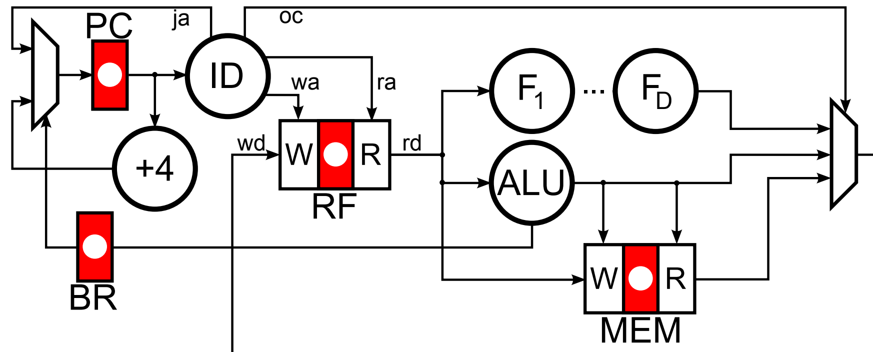


Figure 5.10: DLX initial graph.

5.4 Pipelining Examples

In this section, the method is applied to two microarchitectures, and it is analyzed how different parameters, such as depth of functional units, instruction probabilities or dependency probabilities affect the performance of the system.

The presented algorithms have been implemented inside the MAREX toolkit. Then, they have been tuned using a large set of microarchitectural graphs. In particular, if the algorithm stores only those design points which have an estimated effective cycle time within 1% from ξ_{min}, then the design point with the best effective cycle time is found with a 70% success rate. When the optimal design point is missed, the performance degradation with respect to the optimal point is on average only 3%. The average number of simulations required is 4.125. The success rate of finding the optimal performance design increases to 91% if the solution frontier threshold is set to 5%. Finally, the best design point has always been found with the threshold set to 10%. The average number of simulations are 12.6 and 24.3, respectively. The algorithm scales well up to hundreds of nodes - enough for realistic IP blocks and embedded CPUs.

Furthermore, the presented method has been used to support a practical test case, in which the advantages of elastic systems have been studied using a video decoding engine of the industrial SOC [68]. Section 5.5 provides some information on how the contributions of this thesis were used in this 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

Table 5.1: Delay and area numbers for DLX example using NAND2 with FO3 as unit delay and unit area.

5.4.1 DLX pipeline

This method is first illustrated on a simple microarchitecture similar to a DLX, shown in Fig. 5.10 before pipelining. The execution part of the pipeline has an integer ALU and a long operation F. The instruction decoder ID produces the opcode, *oc*, that goes to the write-back multiplexor and a target instruction address, *ja*, that is taken in function of the previous ALU operation, as stored in the register BR. Table 5.1 shows approximate delays, latency and area of the functional blocks of the example, taking NAND2 with FO3 as unit delay and unit area. In order to obtain these parameters, some of the blocks have been synthesized in a 65nm technology library using commercial tools (ALU, RF, mux2, EB and +4), and the rest of the values have been estimated. EB and mux2 delay and area numbers were taken for single bit units. The delay of bit-vector multiplexors and EBs is assumed to be the one shown in the table, while area is scaled linearly w.r.t. the number of bits. Multiplexors with a fan-in larger than two are assumed to be formed by a tree of 2-input multiplexors.

Even though a conventional DLX pipeline does not typically include this feature, F is considered to be a floating point unit. Its total delay is set to 80, around 5 times the expected cycle time of the design. Then, it may be partitioned in several stages ($F_1, F_2, \ldots F_D$) to allow pipelining. It is assumed that the delay of each stage is $80/D$. Notice that partitioning of F must be provided by the designer *a priori*, it is not performed automatically by the exploration algorithm. However, it is possible to provide several possible partitionings and perform design exploration on each one, thus determining the optimal pipeline depth.

The register file is 64 bits wide, with 16 entries, 1 write and 2 read ports. The total footprint of the RF is 6000 units, (including both cell and wire area). To account for wiring of other blocks, we assume that 40% space is reserved for their wiring. Furthermore, we also need to consider the area overhead of elastic controllers. Based on experiments with multiple design points, a 5% area

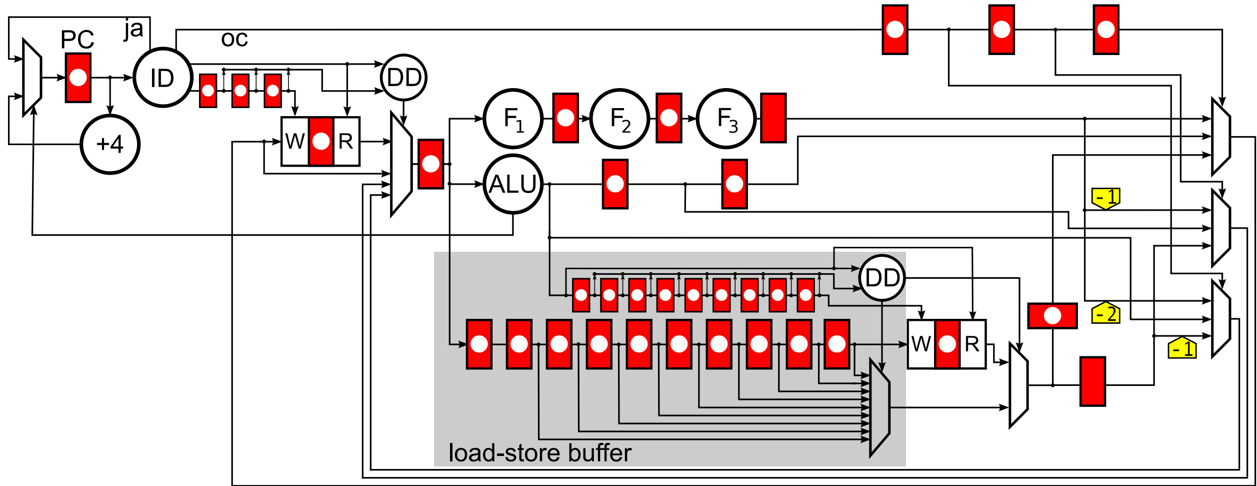


Figure 5.11: Pipelined DLX graph (F divided into 3 blocks. RF has 3 bypasses and M 9).

is reserved for the controllers. Given that $\text{Area}_{\text{Blocks}}$ is the area due to the different combinational blocks plus the area of all the EBs, the total area of the design is $\text{Area}_{\text{RF}} + (\text{Area}_{\text{Blocks}} * 1.05)/0.6$). The area of the initial non-pipelined design shown in Fig. 5.10 is 23284 units.

The area of the memory subsystem is not taken into account (as it is roughly constant regardless of pipelining). It has a parameterized latency of L_{MEM} cycles for the read instruction, which is set to 10 in table 5.1. A latency of 10 clock cycles for read instructions is realistic if we consider it the latency of the L2 cache access. Read operations are considered to be non-blocking, i.e., a new token can be accepted even if the previous one is still in flight. This is modeled by a transition with infinite server semantics in the TGMG model.

Other parameters of the microarchitecture are the data dependency probability in both the register file and the memory ($\gamma_{\text{RF}}, \gamma_{\text{MEM}}$), the probability of a branch to be taken (p_{BR_T}) and the probability of each instruction to be executed ($p_{\text{ALU}} + p_{\text{F}} + p_{\text{LOAD}} + p_{\text{STORE}} + p_{\text{BR}} = 1$). These probabilities are mapped to the early-evaluation multiplexors of the graph.

Design Point Example

Figure 5.11 shows one of the best design points found by the algorithm under the following design parameters: the F unit has been divided into three blocks, the memory data dependency probability is 0.5 ($\gamma_{\text{MEM}} = 0.5$), and register file data dependency probability is 0.2 ($\gamma_{\text{RF}} = 0.2$), the instruction probabilities are: ($p_{\text{ALU}} = 0.35$, $p_{\text{F}} = 0.2$, $p_{\text{LOAD}} = 0.25$, $p_{\text{STORE}} = 0.075$, $p_{\text{BR}} = 0.125$). Finally, the probability of a branch taken is 0.5. These values are based on the experiments found in [75].

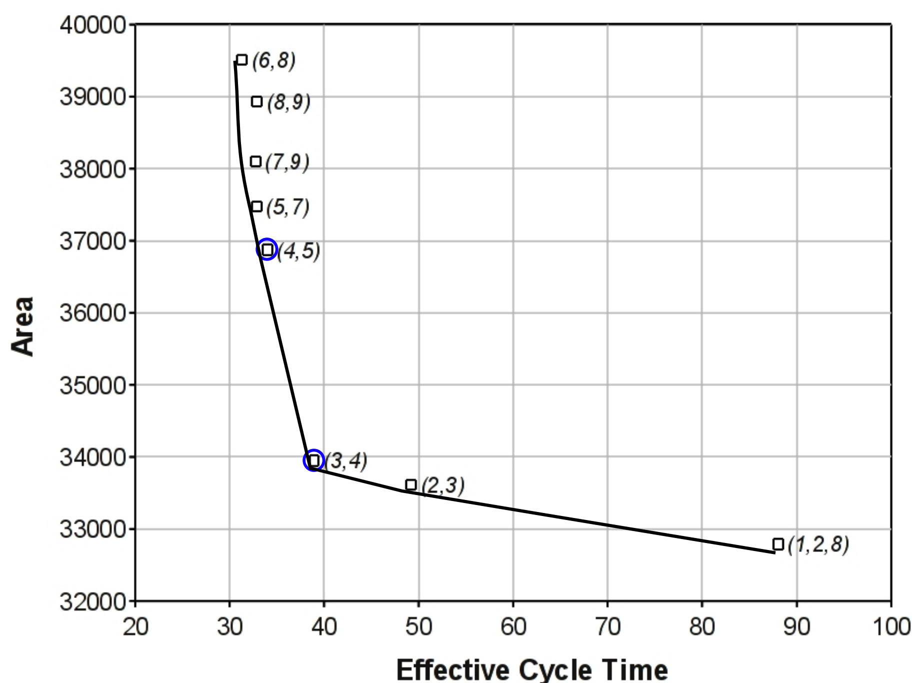


Figure 5.12: Effective cycle time and area of the best pipelined design for different depths of F. (x,y) and (x,y,z) tuples represent the depth of F, the number of bypasses applied to RF and to MEM (z = 9 if omitted).

In Fig. 5.11, the cycle time is 29.817 time units, due to the F_1, F_2 and F_3 functional blocks. 3 bypasses have been applied to RF and then EBs have been retimed to pipeline F. Note that the algorithm inserted an extra bubble at the output of F_3: the reduction in the throughput due to this bubble, is compensated by a larger improvement in the cycle time (without this bubble the critical path would include the delay of the multiplexors after F_3). If F operation had a higher probability, the design without this bubble might be better, since the bubble would have a higher impact on the throughput degradation. Such decisions are made *automatically* based on the expected frequencies of instructions and data dependencies. An alternative way to avoid the bubble after F_3 is to add an extra bypass to RF. An extra bypass is indeed the optimal way to pipeline the design for this particular instance of the parameters. It may not be the case if the data hazard probability was higher.

Optimal Depth

Figure 5.12 shows the effective cycle time and area of the best design point found by the presented method on different depths of F, forming a Pareto-point curve. The first point on the x scale corresponds to a graph where F is a single node, the second one corresponds to a graph where F is partitioned into F_1, F_2, etc. The parameter values are the same ones as in Fig. 5.11. The delay of F is the largest one, and as the number of nodes in which it is partitioned increases, the maximum node delay decreases, and so does the best possible cycle time in the graph. In the best design for the graph in which F is not partitioned ($\mathsf{depth}(\mathsf{F}) = 1$), the effective cycle time is 87.99, the area is 32791, 2 bypasses have been applied to RF and 8 bypasses have been applied to MEM. The cycle time of this first point is 83.15 ($\delta(\mathsf{F}) + \delta(\mathtt{EB})$), and the throughput is 0.94. Only back-to-back data dependencies involving F instructions require stalling of the pipeline.

As depth(F) increases, more bypasses are applied to the register file. The area of the design increases with more bypasses. For $\mathsf{depth}(\mathsf{F}) = 2$, the cycle time is 43.15, the throughput is 0.86 and the effective cycle time is 49.77. Until $\mathsf{depth}(\mathsf{F}) = 6$, the cycle time keeps decreasing because of the deeper pipeline (43.15, 29.81, 23.15, 19.26, 16.48), and so does the throughput because more data hazards must be handled. Overall, the effective cycle time improves with a deeper pipeline until 6 stages are reached.

For $\mathsf{depth}(\mathsf{F}) = 7$, the delay of each stage of F becomes lower than the delay of the ALU ($\delta(\mathsf{F}_i) = 80/7 = 11.4$). Thus, the cycle time cannot be further improved by increasing the depth of F. Thus, the best effective cycle time is achieved with F divided into 6 stages, 8 bypasses applied to RF and 9 to MEM. Design points (4,5) and (3,4) (circled in the figure) for 4 and 3 stages are simpler and overall might deliver a better design compromise.

Different parameters lead to different optimal design points. For example, when $\gamma_{\mathsf{RF}} = 0.5$, bypass paths are selected more often and the throughput is generally worse, since more stalling is needed. However, in this case the optimal depth of F becomes 7 instead of 6. Different instruction probabilities also lead to different optimal design points. If the microarchitecture issues ALU instructions most of the time, the throughput is close to 1 on the designs, since data dependencies can always use forwarding in order to avoid stalls. Besides, since F operations are never selected, less bypasses are applied and recycling is used in order to pipeline F. Because of the store buffer added by bypasses on MEM, memory operations can also be performed with a high throughput. On the other hand, if F instructions are the most common ones, the throughput degrades faster as this instruction always needs to stall the pipeline when a data dependency occurs.

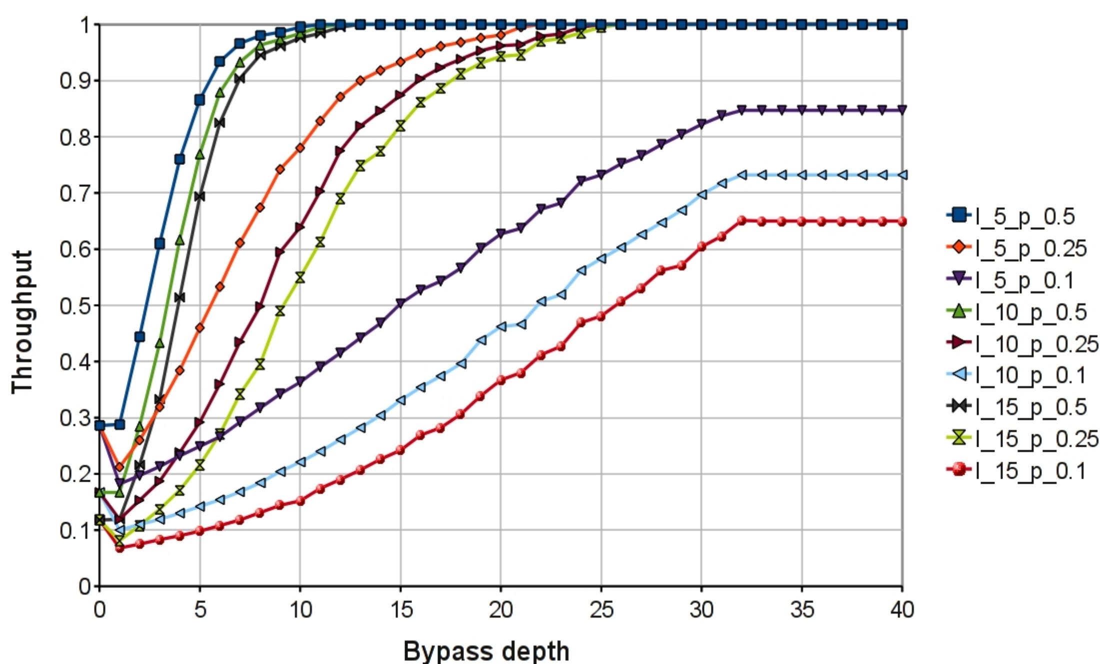


Figure 5.13: Throughput of the memory subsystem given different read latencies and bypass dependency probability. In the legend, l_x_p_y, means $L_{\text{MEM}} = x, \gamma_{\text{MEM}} = y$.

The time needed to run algorithm 3 is usually around 200 seconds for every DLX configuration. The longest exploration took 400 seconds. Parameters such as F depth or memory latency have an impact on the run-time, since more bypasses need to be applied. Most of the exploration time, around 93%, is spent solving RR ILP problems using the CPLEX solver [55]. Simulation time depends on the number of near-optimal points found by the algorithm. If the threshold is set to 1%, the average number of simulations is 4.125, and the average simulation time is 150 seconds. Higher thresholds provide more variety of possible designs but require more time in order to simulate all of them.

Memory subsystem

Bypasses done on RF are useful because the inserted EBs are retimed to pipeline F. On the other hand, memory bypasses are used to hide the memory latency via a *load-store buffer*, as shown in Fig. 5.11. When a read operation starts, if the address being read has been written during the last 9 clock cycles, the early-evaluation multiplexor can get the token from the store buffer without waiting for the read operation. Otherwise, the multiplexor stalls the execution until the token from

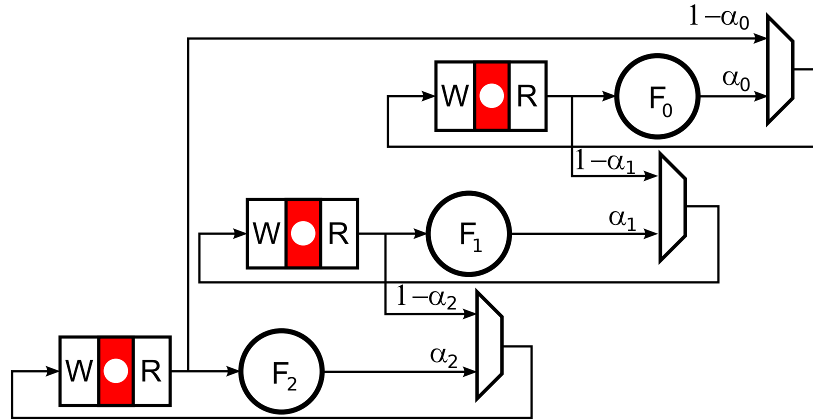


Figure 5.14: Ring of pipelines.

the memory arrives. Such structure can be substituted by a more efficient implementation: an *associative memory*. The algorithm automatically detects the need for a load-store buffer and its optimal size.

Figure 5.13 shows the throughput of the memory subsystem given different latencies of memory reads and different dependency probabilities. The x-axis is the number of bypasses applied. It can be seen that the lines in the figure that have the same bypass probability γ_{MEM} have the same shape, shifted to the right as the latency is larger. Lower memory latencies, larger number of bypasses and larger dependency probabilities affect the throughput positively. If data dependencies are rare, it is not possible to reach a throughput close to 1. Also notice that for the parameters used on the previous experiment ($L_{\mathsf{MEM}} = 10, \gamma_{\mathsf{MEM}} = 0.5$), 9 bypasses give an optimal throughput.

5.4.2 Ring of pipelines

In order to further discuss this method and the implication of different microarchitectural parameters on the performance, a simple parameterized microarchitecture is used in this section.

This microarchitecture is formed by k pipelines, each one consisting of a functional unit F_i and a register file R_i, where $0 \leq i < k$. Each register file R_i receives the output of F_i with probability α_i, otherwise it receives the output of register file $R_{(i-1) \bmod k}$, forming a ring of pipelines. Figure 5.14 shows the microarchitecture for $k = 3$.

Each register file has a bypass probability γ_i, and each functional unit has a delay $\delta(F_i)$. F_i has a depth d_i, and hence it is split into d_i units units $F_{i,0}, \ldots F_{i,d_i-1}$.

This example is useful because it allows to explore a microarchitecture with an arbitrary num-

Par.	Range	Par.	Range
k	$[1,8]$	$\delta(F_{i,j})$	$400/D$
γ	$0, 0.001, 0.01, 0.1$	$\delta(mux)$	$10 + \log_2(\#\text{inp.})$
D	$[2,30]$	$\delta(\text{read}(R_i))$	20
α_i	random(0.2, 0.8)	$\delta(\text{write}(R_i))$	20

Table 5.2: Parameter ranges in the experiments.

ber of register files. Being all functional units simple pipelines with no additional elements, there are no interferences in the results due to other critical paths.

By trying different depths on each pipeline, different γ_i and different k, it can be seen how these parameters affect the performance of the design. The design has been tested setting the depth of each pipeline to the same value. For each $D \in [2,30]$, a graph has been created where $\delta(F_{i,j}) = 400/D$ and $\text{depth}(F_i) = D$. 400 units of delay is a reference number considered the total delay of each F_i. The read and write delay in the register files is fixed to 20 units, and the probability of reading from the pipeline α_i is a random number between 0.2 and 0.8.

In order to test different values of k and γ, each depth has been tested with 1 to 8 register files, and with four different bypass probabilities (0, 0.001, 0.01 and 0.1). All register files have the same bypass probability γ in these tests. Dependency probabilities are set to smaller values compared to DLX experiments because it enables us to explore deeper pipelines. If γ is set to a higher value, the throughput of the pipeline drops to very small values. Table 5.2 lists the parameter ranges used for the experiments.

The delay of each multiplexor, including bypass ones, is set to $10+\log_2(\#\text{inputs})$ units. However, we have found that multiplexor delay and EB delay do not affect the performance of the best pipelining found. This is true unless their delay is set to an unrealistic high number or the multiplexor is located on a critical cycle in a shallow pipeline.

Figure 5.15 shows the effective cycle time of the automatically pipelined designs for different depths, with bypass probability equal to 0.1. The three different curves correspond to different number of pipelines ($k = 2, k = 5, k = 8$). In all cases, the depth with best performance is 20. From 2 to 20, as the depth increases, the delay of each $F_{i,j}$ is smaller, and hence the cycle time is smaller. Even though a deeper pipeline also means a worse throughput, $\gamma = 0.1$ is small enough so that the gain in cycle time overweights the reduction of the throughput.

The delay of the read and write operations of the register file is set to 20. Once the depth is greater than 20 ($D \geq 20$), the cycle time cannot continue to improve, because $\delta(F_{i,j}) = 400/D \leq 20$.

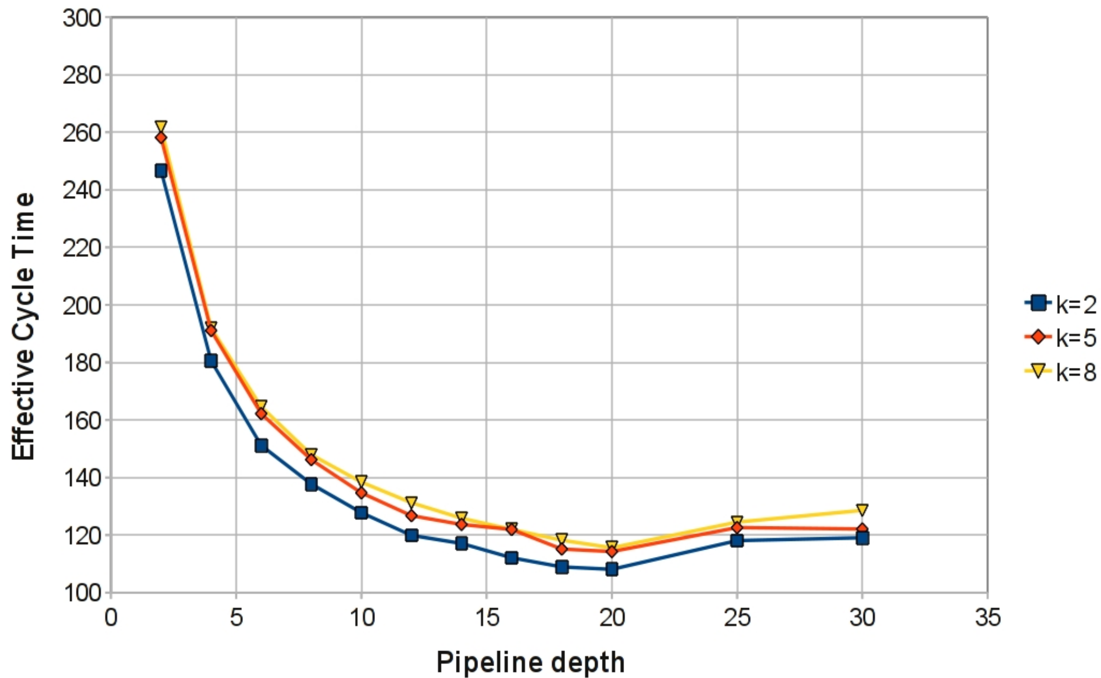


Figure 5.15: Effective cycle time for different pipeline depth for $\gamma = 0.1$.

The read operation of the register file becomes the maximum node delay. At this point, deeper pipelines pay a throughput penalty with no cycle time gain.

The bypass probability most sensitive to k increases is $\gamma = 0.1$. For smaller bypass probabilities, there is a smaller difference between different k's. As the number of data hazards increases, the number of stalls that each pipeline causes to its neighbors is also larger. If there is a small amount of stalls, there is a small impact to the throughput. However, if the number of stalls and the number of pipelines is large, the number of stalls in the ring increases, producing a more significant degradation of the throughput due to inter-pipeline communication.

Figure 5.16 shows the effective cycle time for different depths with different bypass probabilities (γ is written using letter 'p' in the figure) and k set to 5. It can be seen that as data hazard frequency decreases, the number of cycles the pipeline is stalled is smaller, and thus, the performance of the system is better.

In this example, a better performance is always achieved by assigning each functional unit to its own pipeline stage, rather than combining them in pairs. Thus, the cycle time for all the different parameter combinations is $\delta(F_{i,j})$. It is also interesting to note that the throughput in this example

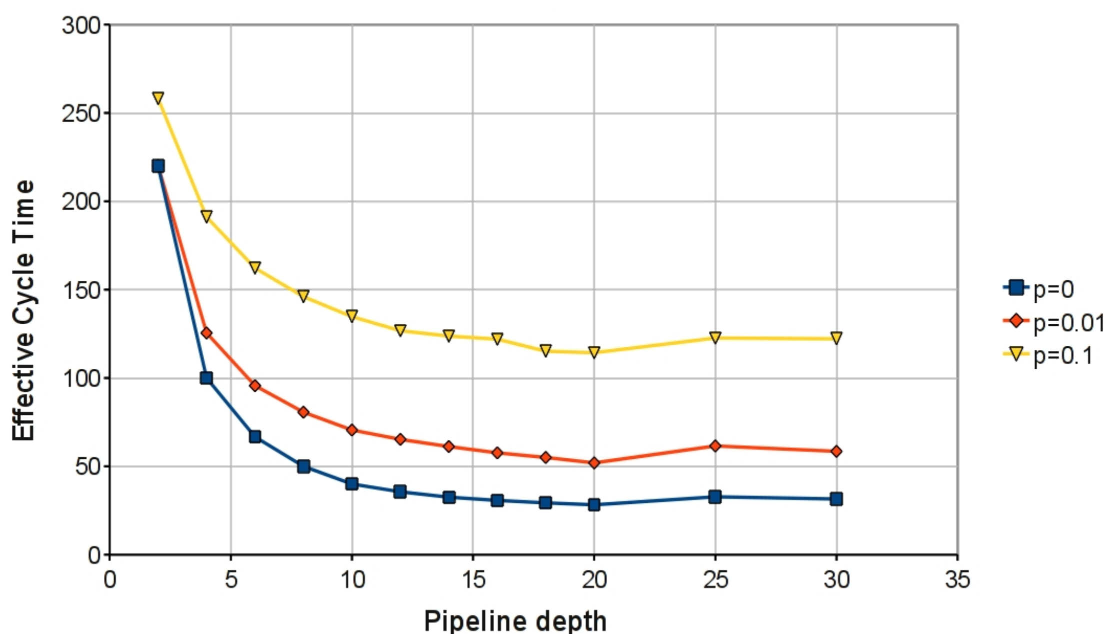


Figure 5.16: Effective cycle time for different pipeline depths for $k = 5$.

γ	D	#byp	$D/\#byp$	cyc	ξ	cyc	ξ
0	20	20	0.850	20	20	40	40
0.001	20	16.2	0.81	20	35	40	46
0.01	20	14.2	0.71	20	51	40	71
0.1	20	6	0.30	20	114	40	136

Table 5.3: Number of bypasses performed in the pipelined design.

is always worse than in DLX examples for similar dependency probabilities. Since there is no short instruction, like ALU in DLX, all data hazards must stall until the needed instruction finishes. No forwarding is possible.

Table 5.3 shows the number of bypasses per register file done for the best design point explored, for each of the bypass probabilities. As expected, bypass transformations are extensively used when data hazards rarely occur. On the other hand, for bypass probability of 0.1, few bypass transformations are done and pipelining is achieved by adding bubbles.

The two sets of columns (cyc, ξ) on table 5.3 correspond to 2 of the Pareto points found by RR for the indicated number of bypasses. As γ increases and more bubbles are needed in order to reach the minimum possible cycle time (20), the effective cycle time distance between using a

k	D	Min	Q1	Mean	Med	Q3	Max
*	*	44	229	612	452	921	1900
2	*	44	151	216	189	318	420
2	(0,10)	44	89	148	151	185	383
2	(10,20)	129	169	246	264	331	380
5	*	92	405	581	553	863	1098
5	(0,10)	92	230	367	405	547	676
5	(10,20)	339	510	696	693	919	1098
8	*	160	776	1038	978	1345	1900
8	(0,10)	160	373	679	799	978	1197
8	(10,20)	640	953	1265	1218	1682	1871

Table 5.4: Run-time of the whole exploration method (including simulations) for different number of pipelines (k) and different depth ranges (D) in seconds. The minimum, maximum, mean, medium and quartiles are shown for each range.

cycle time of 20 and using a cycle time of 40 is reduced. However, a cycle time of 20 is still better for all cases.

Table 5.4 shows run-times of the exploration method (including simulations) for different number of pipelines and different depths. The whole exploration and simulation run-time for experiments on this microarchitecture takes from less than a minute for small graphs to half an hour for larger graphs. Both the number of pipelines and the pipeline depth modify the run-time since they affect the number of nodes and edges of the graph, and thus, the number of variables on the ILP problems. For example, a graph with two pipelines and depth 20, after 5 bypasses per pipeline has 66 nodes and 78 edges, while the same graph with 8 pipelines instead of 2, has 264 nodes and 312 edges. Besides, bypass transformations add extra nodes and edges in a graph. For example, a graph with 2 pipelines and depth 6, has 32 nodes and 38 edges after one bypass applied per pipeline, and 38 nodes and 50 edges after 5 bypasses applied per pipeline.

The maximum run-time is 1900 seconds. However, 75% of the runs need less than 1000 seconds to complete (as shown by Q3 column). Only deep pipelines with $k = 8$ require that much time. Approximately half of the runs needed less than 10 minutes and more than 25% of the runs complete in less than 5 minutes (as shown by Q1 column). Overall, the table shows how both k and D have a direct impact on the run-time.

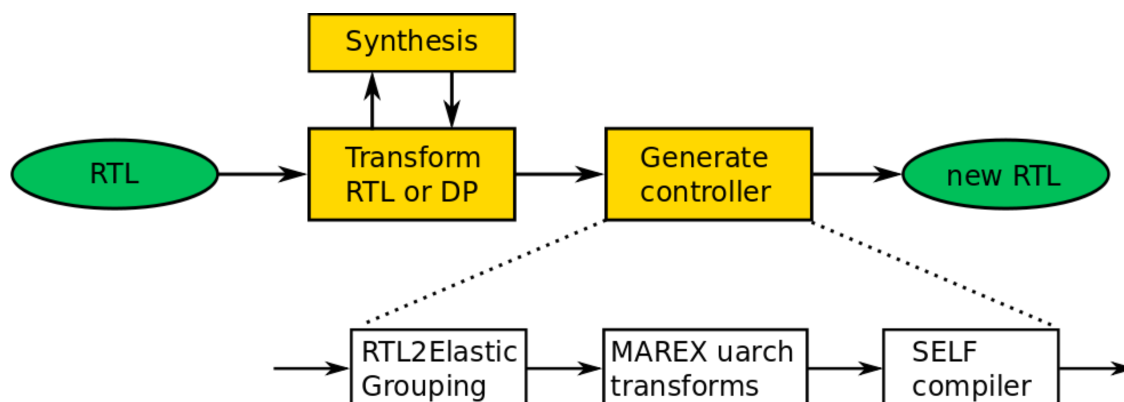


Figure 5.17: Flow for video decoding engine design experiment.

5.5 Video Decoding Engine

A practical test case of optimization using elastic systems is presented in [68]. Among other optimization techniques, some of the procedures presented in this chapter were applied in this paper in order to optimize the critical core of the video decoding engine.

When redesigning an existing synchronous design into a synchronous elastic design (like in this experiment), it is natural to separate datapath synthesis and optimization from controller synthesis and optimization, although it must still be possible to connect them afterwards.

The design flow used for the experiment is shown in Fig. 5.17. Starting from the existing RTL, a set of design transformations are applied, such as little care and critical core separation, adding precomputation, bubble insertion, retiming, etc. These changes are applied manually on the RTL (more details are explained in [68]). To check how these transformations affect timing, area, and power of the design, we can run a commercial synthesis flow (either logic synthesis only or both logic and physical synthesis). These synthesis checks provide feedback to which transformations should be accepted and which should be conducted next.

After the desired timing/area/power trade-off is achieved on the datapath, the elastic controller is generated and merged with the datapath (where registers have been replaced by elastic buffers), producing the new elasticized RTL. This complete new elastic RTL is again synthesized and compared to the original design.

The controller generation is automated and follows three steps: RTL2Elastic, MAREX, and SELF compiler. RTL2Elastic is a tool that can analyze an RTL design and produce a microarchitectural graph, including early-evaluation multiplexors. The RTL sequential components are grouped

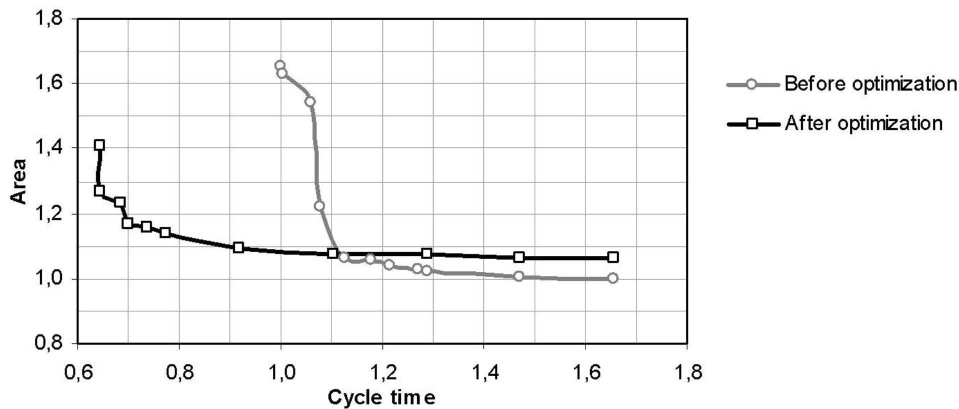


Figure 5.18: Area-performance trade-off of the original and the optimized decoder.

into elastic islands, i.e., several registers are controlled by a single EB controller, as decided by the user of the tool. Finally, MAREX reads the microarchitectural graph from RTL2Elastic and produces a Verilog controller of the elastic design using SELF compiler.

MAREX also provides all the elastic transformations and the automatic exploration engine, and it can be used to guide the RTL optimization. After back-annotating the graph with delays obtained from synthesis, the tool could automatically suggest elastic transformations, such as retiming moves or places where a bubble could be added to enhance the performance. Transformations could be evaluated quickly before trying them on the RTL.

Furthermore, starting from the original microarchitectural graph, the same transformations that had been applied on the RTL could be applied on the graph, checking that the results matched. Another decision point was to figure out which of the multiplexors had to be implemented as early-evaluation ones, and which ones should be implemented using simpler controllers. Similarly, few channels actually needed to implement token counterflow. Most of them could be implemented using passive anti-tokens, significantly simplifying the final controller.

Figure 5.18 shows the performance/area trade-off curves between the original synchronous version of CABAC and the optimized elastic version. These are normalized results after logic synthesis using a 32 nm library. Given equal area, the cycle time improvement is up to 40%, depending on which cycle time is chosen. Testbench simulations provided a throughput of 0.91 in the final elastic design. Thus, the effective cycle time speed-up can be up to 35%.

5.6 Conclusion

This chapter has presented a method for automatic pipelining of elastic systems. This method takes advantage of optimization techniques available for elastic systems, such as retiming and recycling, and extends them by exploring different bypass configurations on memory elements of the design.

After applying a number of bypasses to each memory following an heuristic algorithm, and enabling the corresponding forwarding paths to resolve data hazards, the retiming and recycling optimization method is called. This method solves a mixed integer linear programming problem which models most of the elastic transformations. RR returns a set of non-dominated pipeline configurations with different throughput versus cycle time trade-offs. Since the throughput has been estimated inside RR using analysis methods that are not exact, the most promising design points are simulated at the end of the exploration in order to obtain their actual throughput.

The presented exploration engine can find hundreds or even thousands of different pipelines with the same functionality as the original input microarchitecture. The exploration is guided by the expected instruction frequencies and the expected data dependencies. This way, a pipeline with a near-optimal performance or a good performance/area trade-off for the expected workload can be achieved. The engine can also output a set of Pareto points with different trade-offs between clock cycle, throughput, area and power such that a designer or an architect can select the best suited for a given application.

Using the presented method with different partitions in the functional modules it is possible to quickly analyze the optimal pipeline depth for a given microarchitecture, conducting pipelining studies similar to what is proposed in [72, 79].

The method has been applied to several pipeline designs, showing how different parameters, such as pipeline depth or data hazard probabilities, affect the performance of the system and the shape of the optimal pipeline.

Several extensions can be added to the presented method. Currently, each node is assigned a latency, delay and area. However, it would be possible to assign several possible configurations per node, each one with a different trade-off between latency (either fixed or variable), combinational delay and area. For example, an ALU could have three possible configurations, the first one with the largest delay, the smallest area and latency of one clock cycle, the second configuration with a smaller delay, larger area and latency of one, and the third configuration with the smallest delay due to variable latency, but also the largest area. All these possible options could be encoded inside the RR optimization method, which would be able to pick the best one in function of the criticality

of each node.

It is also possible to try to apply speculation, as presented in Chapter 4, to some of the configurations found by RR if the throughput is found to be limited by a cycle going through the control input of an early-evaluation node, The user should provide a set of pre-defined schedulers that must be able to predict some of the choices made inside the system in order to be able to apply speculation automatically.

Chapter 6

Symbolic Performance Analysis

Elastic systems, either synchronous or asynchronous, can be optimized for the average-case performance when they have units with early evaluation or variable latency. The performance evaluation of such systems using analytical methods is a complex problem and may become a bottleneck when conducting an extensive exploration of different architectural configurations.

This chapter proposes an analytical method for performance evaluation using symbolic expressions. The presented method combines timing simulations and symbolic algebraic expressions, and it supports early evaluation and variable-latency units. First, an exact version of this method with an exponential worst case complexity is presented. Then, a fast approximate version of this method is presented that prunes away low-impact event correlations and computes a *lower bound* of the system throughput.

A fast method to estimate the throughput can improve the exploration engine presented in Chapter 5. A fast upper bound method already exists in the literature [85, 86], and a fast lower bound method helps to restrict the actual throughput and to potentially reduce the number of simulations needed at the end of the exploration. The contribution presented in this chapter has been published in [63].

6.1 Introduction

Figure 6.1(a) shows a timed marked graph with early evaluation and variable-delay transitions (see Section 2.1 for an introduction to timed marked graphs). This chapter will introduce *Timed multi-Guarded marked Graphs with Variable-delay* (TGVs), an extension to timed marked graphs which

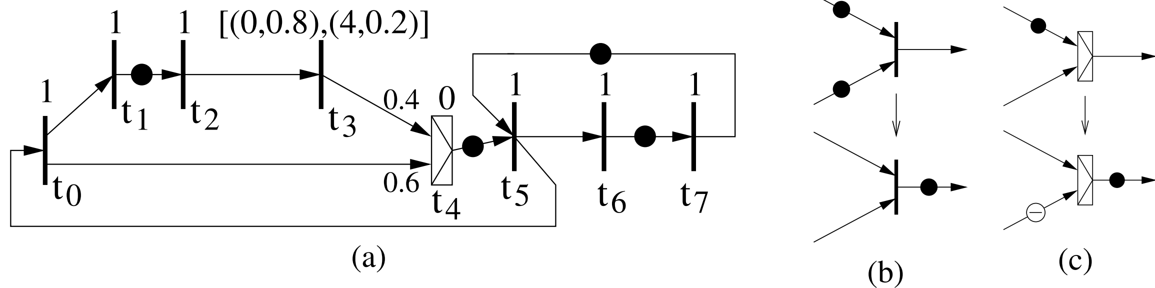


Figure 6.1: (a) A TGV with early evaluation and variable-delay, (b) AND-causality firing, (c) early-evaluation firing.

can model early-enabling and variable-delay transitions. A formal definition of TGVs is presented in Section 6.2.1. In Fig. 6.1(a), transitions of the marked graph are represented as thick vertical lines, places as the edges between transitions, and tokens as dots over the edges.

Figure 6.1(b) shows conventional AND-causality firing in a marked graph, and Fig. 6.1(c) shows firing with early evaluation, which adds an anti-token into the empty input places of the transition. In early-evaluation transitions, the selection of the required branch in the datapath is abstracted by using non-deterministic choices. After each firing, the transition chooses which of the inputs will be required for the next firing. It can be assumed that it will require only one of the inputs to be valid without loosing generality [85]. Early-evaluation transitions are drawn as special boxes, like transition t_4 in Fig. 6.1(a).

Each input of an early-evaluation transition is assigned a probability that will be used for performance analysis. After each firing, early-evaluation transitions choose which input will be required for the next firing using these probabilities. In Fig. 6.1(a), the input from t_3 is selected with probability 0.4, and the input from t_0 is selected with probability 0.6.

Each transition is assigned a delay, drawn on top of the transition in Fig. 6.1(a). Transition t_1 has delay 1. This means that once it is enabled, it needs 1 time unit to fire. It is possible for a transition to have several possible delays, modeling variable-latency units. Each possible delay is assigned a probability. For example, transition t_3 will need 0 time units to fire with probability 0.8 and 4 time units with probability 0.2. It may represent an ALU with a short operation that can be computed combinationally and a long operation that needs 4 clock cycles. The delay of variable-delay transitions is represented as a set of delay-probability pairs: $[(0,0.8),(4,0.2)]$.

The throughput of a TGV is defined as the number of times a transition can fire on average per time unit. The only work that has studied the performance analysis of systems with early

evaluation so far is [85, 86]. The paper proposes to use linear programming (LP) in order to obtain an upper bound of the throughput. It does not handle variable-delay transitions directly, but it is mentioned that they can be modeled by using the average delay of the transition (for example 0.8 for t_3).

This chapter proposes an approach to compute the throughput by extracting the probability distribution of the firing times of transitions. Instead of directly deriving the probability distribution for each firing time, the presented method derives symbolic expressions, that can correctly capture and manipulate correlations between timing events. Then, the expressions are evaluated to obtain the firing times of transitions. Finally, the throughput is computed by extracting the average separation between firing times of the same transition. For performance purposes, some correlations may be ignored when evaluating the symbolic expressions. It will be shown that a lower bound of the throughput is obtained when correlations are ignored.

The throughput of the system in Fig. 6.1(a) is 0.397, obtained by simulating it. However, the linear programming upper bound method measures 0.449. Consider t_2 in Fig. 6.1(a). Since it is enabled at the beginning of the simulation and it has delay 1, its first firing occurs at time 1 ($f(t_{2,0}) = 1$, where $f(t_{i,j})$ means the time of the j-th firing of transition t_i). Similarly, t_5 and t_7 are initially enabled ($f(t_{5,0}) = 1, f(t_{7,0}) = 1$).

Then, t_0, t_3 and t_6 are enabled. Their firing times are obtained by adding their delays to their arrival times: $f(t_{0,0}) = f(t_{5,0}) + 1 = 2$, and $f(t_{6,0}) = f(t_{5,0}) + 1 = 2$. Since t_3 is a variable-delay transition, the firing time becomes probabilistic. When creating symbolic expressions for firing times, variable delays are modeled using variables. As it will be shown, these variables allow to detect correlations due to reconvergent paths. In some cases, these correlations can be simplified away by manipulating the symbolic expression. For this simple example, the firing time of t_3 is 0 with probability 0.8 and 4 with probability 0.2, and it is captured in variable x_0:

$$\begin{aligned} x_0 = &\quad [(0, 0.8), (4, 0.2)] \\ f(t_{3,0}) = &\quad f(t_{2,0}) + x_0 \end{aligned}$$

Transition t_4 has an early-evaluation enabling function. When it chooses the lower input, which happens with 0.6 probability, its arrival time is the firing time of $t_{0,0}$. On the other side, when it chooses the upper input, its arrival time is the firing time of $t_{3,0}$. We can express this firing time introducing an early-evaluation operation ($\vee$) in the symbolic expressions:

$$f(t_{4,0}) = f(t_{0,0})^{0.6} \vee f(t_{3,0})^{0.4}$$

When a transition has more than one input and it is not early evaluated, the maximum of the firing times of the inputs must be computed in order to obtain the arrival time of the transition. The inputs of t_5 are t_4 and t_7. Hence, the arrival time for $f(t_{5,1})$ is the maximum of $t_{4,0}$ and $t_{7,0}$. Every time a maximum is created, it is checked whether some of the operands are redundant. In this case, $f(t_{7,0}) = 1$, and we can derive that $f(t_{4,0}) \geq 1$. Thus, $f(t_{7,0})$ can be discarded:

$$\begin{aligned} f(t_{5,1}) &= 1 + \max(f(t_{4,0}), f(t_{7,0})) \\ &= 1 + f(t_{4,0}) \end{aligned}$$

Once the symbolic expressions have been built and simplified, they can be evaluated to obtain the average firing time, $\overline{f}(t)$, of each transition. The average firing time of a transition is the mean of the possible delays considering their probabilities.

$$\begin{aligned} f(t_{3,0}) &= f(t_{2,0}) + x_0 \\ &= 1 + [(0, 0.8), (4, 0.2)] \\ &= [(1, 0.8), (5, 0.2)] \\ \overline{f}(t_{3,0}) &= 1 \cdot 0.8 + 5 \cdot 0.2 = 1.8 \end{aligned}$$

$$\begin{aligned} f(t_{4,0}) &= f(t_{0,0})^{0.6} \vee f(t_{3,0})^{0.4} \\ &= 0.6 \cdot 2 + 0.4 \cdot [(1, 0.8), (5, 0.2)] \\ &= [(1, 0.32), (2, 0.6), (5, 0.08)] \\ \overline{f}(t_{4,0}) &= 1 \cdot 0.32 + 2 \cdot 0.6 + 5 \cdot 0.08 = 1.92 \end{aligned}$$

$$\begin{aligned} f(t_{5,1}) &= 1 + f(t_{4,0}) \\ &= 1 + [(1, 0.32), (2, 0.6), (5, 0.08)] \\ &= [(2, 0.32), (3, 0.6), (6, 0.08)] \\ \overline{f}(t_{5,1}) &= 2 \cdot 0.32 + 3 \cdot 0.6 + 6 \cdot 0.08 = 1.92 \end{aligned}$$

Algorithm 4 shows the top level algorithm to compute the throughput of an elastic system with early evaluation and variable-latency units. First, an *unfolding* [61, 107, 119] of the marked graph is created. An unfolding is an acyclic marked graph which "simulates" the behavior of the original marked graph for a number of periods. Each transition of the unfolding corresponds to a firing event of the original marked graph. For example, Fig. 6.2 shows a marked graph which is an unfolding of 2 periods of the marked graph in Fig. 6.1(a), meaning it simulates 2 firings of each

Algorithm 4: Overview of the top level algorithm.

Input: G : timed marked graph, K : $\mathbb{N}$
Output: The throughput of G

```
U = CreateEmptyUnfolding()
E = CreateEmptyExpressionCollection()
done = false
avgf = 0
while ¬done do
    AddPeriods(G, U, K)
    for each new transition t ∈ U do
        BuildSymbolicExpression(E, U, t)
    EvaluateExpressions(E)
    avgf_new = max{AvgFiring(t) | ∀t ∈ G}
    done = avgf + ε ≥ avgf_new
    avgf = max(avgf, avgf_new)
return 1/avgf
```

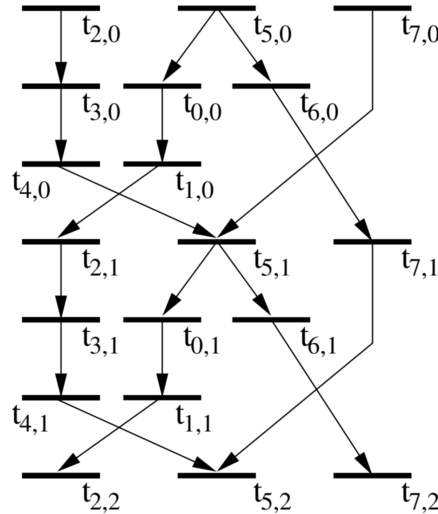


Figure 6.2: 2-period unfolding of the marked graph shown in Fig. 6.1(a).

transition.

The algorithm initially builds an unfolding of K periods. The selected K is discussed in Section 6.4. Then, a symbolic expression is built to represent the firing time of each transition in the unfolding. Each time a maximum or early-evaluation operator is added, some simplification steps are attempted using max-plus algebra properties in order to optimize the size of the expressions

(and hence the efficiency of the whole method). Finally, the expressions are evaluated to find the firing time of each transition.

In Section 6.2, it is shown that the throughput can be found by computing the inverse of the maximum, among the transitions of the marked graph, of the average time distance between consecutive firings in an unfolding which is long enough. After evaluating the expressions, this maximum is computed. While the throughput computed does not converge, more periods are added to the unfolding incrementally. Section 6.4 shows that convergence always occurs and relates it to the accuracy of the throughput calculation.

For example, the following table shows the firing times of t_2 for 11 periods. Furthermore, the time distance between firings and the average firing time distance are also shown. The bottom rows of the table show the average firing time when the initial events are ignored:

$$Avg(j) = \{\overline{f}(t_{2,i}) - \overline{f}(t_{2,j}))/(i - j), \forall i > j\}$$

$Transition$	$t_{2,0}$	$t_{2,1}$	$t_{2,2}$	$t_{2,3}$	$t_{2,4}$	$t_{2,5}$	$t_{2,6}$	$t_{2,7}$	$t_{2,8}$	$t_{2,9}$	$t_{2,10}$
$\overline{f}(t_{2,i})$	1	4	5.92	8.432	10.864	13.3688	15.8787	18.3958	20.9153	23.4369	25.9596
Time dist.	–	3	1.92	2.512	2.432	2.5048	2.5099	2.5171	2.5195	2.5216	2.5227
Avg.(0)	–	3	2.46	2.4773	2.466	2.4738	2.4798	2.4851	2.4894	2.493	2.496
Avg.(1)	–	–	1.92	2.216	2.288	2.3422	2.3757	2.3993	2.4165	2.4296	2.44
Avg.(2)	–	–	–	2.512	2.472	2.4829	2.4897	2.4952	2.4992	2.5024	2.505
Avg.(3)	–	–	–	–	2.432	2.4684	2.4822	2.491	2.4967	2.5008	2.5039
Avg.(4)	–	–	–	–	–	2.5048	2.5073	2.5106	2.5128	2.5146	2.5159
Avg.(5)	–	–	–	–	–	–	2.5099	2.5135	2.5155	2.517	2.5182
Avg.(6)	–	–	–	–	–	–	–	2.5171	2.5183	2.5194	2.5202
Avg.(7)	–	–	–	–	–	–	–	–	2.5195	2.5206	2.5213
Avg.(8)	–	–	–	–	–	–	–	–	–	2.5216	2.5221
Avg.(9)	–	–	–	–	–	–	–	–	–	–	2.5227

The separation between two consecutive events stabilizes around 2.52, and the average distance between events would also converge to 2.52 if more firings where shown. As shown in the table, ignoring the initial firing times makes the average firing time distance converge faster, since only the steady state of the marked graph is considered. By definition, the throughput is the inverse of the average separation between firings of the same transition. Therefore, $1/2.52 = 0.3968$, which

is similar to the throughput obtained by simulating during 10000 cycles, 0.3970.

The rest of this chapter is structured as follows. First, Section 6.2 defines the problem to be solved and other important concepts, such as timed marked graphs that can model variable-delay transitions and early evaluation. This section also relates the throughput to the average time distance between firings of a transition. Then, Section 6.3 defines a max-plus algebra extended with early evaluation that will be used to model the firing time of each transition. It also shows how to build max-plus algebra expressions from an unfolding and which simplification steps are performed to minimize the size of the expressions. Two methods to evaluate the symbolic expressions are presented: an exact method with exponential time cost, A_exact, and an upper bound method with linear time cost, A_prune. The complexity of each method is discussed. Next, Section 6.4 discusses the convergence of both algorithms. Finally, Section 6.5 shows some results and Section 6.6 discusses a possible simplification method for symbolic expressions using SAT-modulo theories (SMT), which was not used due to performance reasons.

6.2 Problem Formulation

This section defines the problem to be solved and some basic concepts needed to understand the solution. The reader is assumed to be familiar with basic Petri nets concepts. See Section 2.1 for some basic definitions and read [115] for a tutorial.

The objective of this work is to determine the throughput of an elastic system with early evaluation and variable-latency units by analyzing a symbolic timing simulation of this elastic system. First, an extension of marked graphs that allow to model the kind of elastic systems under analysis, *Timed Multi-Guarded Marked Graph with Variable-Delay*, are formally defined. Next, unfoldings are presented. In this work, unfoldings are used to perform a finite timing simulation of a marked graph. Finally, the throughput of an elastic system is defined and it is related to the average time distance between consecutive firings of a transition in a marked graph. Thus, it is shown how the throughput can be obtained by computing the firing times of an unfolding.

6.2.1 Multi-guarded Marked Graph

Definition 1 (TGV) *A Timed Multi-Guarded Marked Graph with Variable-Delay (TGV) is a tuple* $N = \langle P, T, Pre, Post, G, \mathbf{m}_0, \delta, \alpha \rangle$ *where:*

- P is a finite set of places, and T is a finite set of transitions.

- $Pre : P \times T \to \mathbb{N} \cup \{0\}$ and $Post : P \times T \to \mathbb{N} \cup \{0\}$ are the *pre* and *post* incidence functions that specify the arc weights. The incidence matrix of the net is $\mathbb{C} = Post - Pre$.
- The *preset* and *postset* of a node $x \in P \cup T$ are denoted as ${}^{\bullet}x$ and $x^{\bullet}$. The Petri net is a marked graph:

$$\forall p \in P, |{}^{\bullet}p| = |p^{\bullet}| = 1.$$

- $\mathbf{m}_0 : P \to \mathbb{Z}$ assigns an initial number of tokens to each place (the initial marking).
- $G : T \to 2^{2^P}$ assigns a set of guards to every transition. The following conditions must be satisfied:

a)	$\forall g \in G(t)\ \ g \subseteq {}^{\bullet}t$.	Each guard is a sub-set of the input places of the transition.
b)	$\bigcup_{g \in G(t)}\ g = {}^{\bullet}t$.	Each input place of the transition is at least in one guard.

- $\alpha : G \to \mathbb{R}^+$ assigns a strictly positive probability to each guard such that for every multi-guarded transition t,

$$\sum_{g \in G(t)} \alpha(g) = 1.$$

- $\delta : T \to \{(d, p)\}$ assigns a list of delay-probability pairs to every transition. It must hold that $d_i \in \mathbb{R}^+ \cup \{0\}$ and for every t,

$$\sum_{(d,p) \in \delta(t)} p = 1.$$

This definition is an extension of the definition of a Timed Multi-Guarded Marked Graph in [85]. The difference is that each transition is assigned a list of possible delays, instead of a single real delay. The delay of each transition t, described by a set of pairs delay-probability $\{(d, p)\}$, represents the number of time units (clock cycles in a latency-insensitive design) that t needs to fire a token. If the delay is equal to 0, it means the transition fires immediately, acting like a combinational circuit. The average delay of a set of delay-probability pairs is the mean of the distribution.

$$\overline{\delta}(t) = \sum_{(d,p)\, \in\, \delta(t)} d \cdot p \tag{6.1}$$

The guards of transitions model *early evaluation*. A regular transition has a single guard corresponding to its inputs $G(t) = \{{}^{\bullet}t\}$. On the other hand, a multiplexor with two possible input places, p_0 and p_1, will have the guards $G(t) = \{\{p_0\}, \{p_1\}\}$. Without losing expressive power, it is assumed

that all early-evaluation transitions have one guard for each input [85]. An early-evaluation transition with input places $p_0, \ldots p_n$ will have n guards, $\{\{p_0\}, \ldots \{p_n\}\}$. For example, t_4 from Fig. 6.1(a) has two guards, one for the edge (t_0, t_4) and one for the edge (t_3, t_4). Each guard is assigned a probability, $\alpha(g)$, used on simulations and performance analysis to decide how often it is selected.

Note that the marking of a place can be negative, modeling anti-token counters. Negative tokens appear when an early-evaluation transition fires and one of the input places does not have a positive token, as shown in Fig. 6.1(c).

Although other marked graph models of elastic designs include backpressure arcs to describe buffer capacities, we choose not to include them for simplicity. This means it is assumed that buffers will always have enough capacity. It would be easy to extend TGVs to include such backpressure arcs.

Definition 2 (Firing semantics) *The dynamic behavior of a TGV system is determined by its firing rules. The execution of a transition t can be described as follows:*

- *Guard selection.* A guard $g(t) \in G(t)$ is selected non-deterministically. Once it is selected, it cannot change until t fires.
- *Delay selection.* A delay $d \in \delta(t)$ is selected non-deterministically. Once it is selected, it cannot change until t fires.
- *Enabling.* Transition t becomes enabled if every place $p \in g(t)$ is positively marked.
- *Firing.* Given a marking $\mathbf{m}$, an enabled transition t can fire leading to a new marking $\mathbf{m'}$ such that $\mathbf{m'} = \mathbf{m} + \mathbb{C}(P, t)$, where $\mathbb{C}(P, t)$ is the column of $\mathbb{C}$ corresponding to t. The firing is performed right after d time units starting from the enabling of the transition.
- *Single-server semantics.* No multiple-instances of the same transition can fire simultaneously. A guard selection is produced for each transition firing, and a transition cannot be enabled again while the previous firing has not completed.

The advantage of this model is that it can precisely model *variable-latency units*. Since the possible latencies are known, instead of just their average, it is possible to take into account correlations between consequent firings of the same transition. For example, if the i-th firing of transition t has a delay of 3 with probability 0.8, then the $(i+1)$-th firing will not be enabled during these 3 time units with probability 0.8. Most properties that apply to the TGMGs defined in [85] also

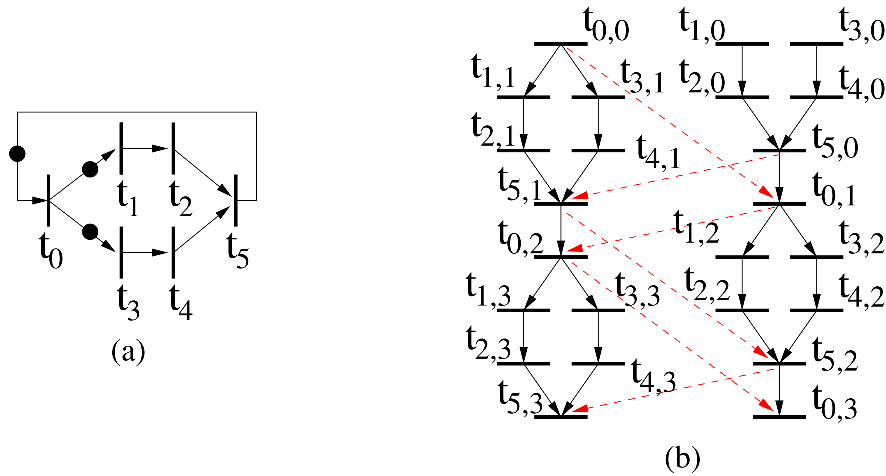


Figure 6.3: (a) example TGV, (b) 4-period unfolding of Fig. 6.3(a), $t_{k,i}$ stands for the i-th instantiation of transition t_k. The dashed edges enforce single-server semantics for t_5 and t_0, the rest of single-server semantics edges are not shown for simplicity.

apply to TGVs, since they are two different timing versions of the *Multi-Guarded Marked Graphs* defined in [85]. For simplicity, we restrict ourselves to bounded strongly-connected graphs.

6.2.2 Unfoldings

An *unfolding* of a marked graph [61, 107, 119] is an acyclic marked graph, where each transition corresponds to the firing of one of the transitions in the original marked graph. An unfolding can be divided into *periods*. The i-th period contains the i-th instantiation of each event.

An unfolding of a Petri net $P = \{P, T, F, M_0\}$ is a tuple $U = (B, E, F', \pi)$ where (B, E, F') is an occurrence net (a subclass of Petri net which is bounded, acyclic, has no auto-conflicts and no backward branching) and π is a morphism which assigns transitions and arcs of U to transitions and arcs of P respectively.

Each transition $b \in B$ is a tuple $b = (t, i)$, where $t \in T$ is the corresponding transition in the Petri net ($\pi(b) = t$) and i is a natural number which indicates the firing count of t for the prefix of the unfolding developed up to (t, i). Similarly, the arcs (aka events) in the unfolding reproduce the structure of the original Petri net while being consistent with the instantiation counters of the transitions.

For example, the marked graph in Fig. 6.3(b) is an unfolding of four periods from the marked

graph in Fig. 6.3(a)[1]. Initially, the transitions t_1, t_3, t_0 from the marked graph are enabled. Thus, in the unfolding they are the first ones to fire ($t_{1,0}, t_{3,0}, t_{0,0}$), where $t_{k,i}$ means the i-th firing of transition t_k ($\pi(t_{k,i}) = t_k$). The firing of t_0 enables the second firing of t_1 and t_3 ($t_{1,1}, t_{3,1}$), and so on.

6.2.3 Single-Server Semantics

In order to model elastic systems properly, the transitions of a TGV have single-server semantics, i.e., each firing can only start once the previous firing finished. In order to ensure *single-server semantics*, some extra edges must be added to the unfolding. These edges make sure that the transition is not re-entrant, i.e. the $(i + 1)$-th occurrence of a transition is not enabled before the i-th occurrence has been fired. For each transition t, a single-server semantic place must be added from t_i to t_{i+1}, for all i (see single-server semantics for t_0 and t_5 in Fig. 6.3(b)).

Adding single-server semantics places for all transitions can significantly increase the number of edges in the unfolding, and hence the run-time of all the algorithms working on this unfolding. Fortunately, it is not necessary to add them for all transitions. In a synchronous elastic system, where all the delays of the TGV are natural numbers, it is sufficient to add single-server semantics places on transitions with delay greater than 1 and for transitions which have a fan-in greater than one, since they are the only transitions that can introduce backpressure (the first ones because they have not finished computing the previous token, and the second ones because some of the inputs are not available yet). Furthermore, single-server semantics edges must also be added between the first and second instantiation (between $t_{i,0}$ and $t_{i,1}$). Otherwise, transitions $t_{1,0}$ and $t_{1,1}$ in Fig. 6.3(b) could have the same firing time if the delay of t_0 was 0.

In Fig. 6.3(b), single-server semantics edges are only needed for t_5 if none of the other transitions is variable delay. It is easy to show that the rest of single-server semantics places become redundant. For example, the single-server semantics edge between $t_{0,0}$ and $t_{0,1}$ already ensures single-server semantics between between $t_{1,1}$ and $t_{1,2}$, and single-server semantics between $t_{1,2}$ and $t_{1,3}$ are ensured by the edge between $t_{5,0}$ and $t_{5,1}$.

6.2.4 Throughput

Definition 3 (Steady state throughput) *The steady state throughput of a transition t, $\Theta(t)$, of a*

[1]The dashed edges enforce single-server semantics for transition t_5 and t_0, see Section 6.2.3 for further details

TGV is defined as:

$$\Theta(t) = \lim_{\tau\to\infty} \frac{\sigma(t,\tau)}{\tau} \tag{6.2}$$

where τ represents time and $\sigma(t,\tau)$ is the firing count of t at time τ, i.e., it indicates how many times t has fired at time τ.

The firing process is *weakly ergodic* if the limit in equation 6.2 exists [30]. It can be shown that this limit does exist for a TGV and that it is the same for all transitions [85].

6.2.5 Timing Simulation of a TGV

The *throughput* of an elastic system can be determined by computing the *firing time* or *occurrence time* of each transition in an unfolding which is long enough. In [118], the occurrence time of the events of an unfolding is computed in order to obtain the cycle time of an asynchronous circuit with only AND-causality transitions. The firing time of a transition becomes a probability distribution if there is early evaluation or variable-latency in the system.

Definition 4 (Firing time of a transition of an unfolding) *The probability distribution for the firing time of a transition t in an unfolded TGV can be defined as follows:*

$$\mathbb{P}(f(t) \equiv X) = \mathbb{P}(\delta(t) + \max\{f(t') \mid t' \in {}^\bullet G(t)\} \equiv X) \tag{6.3}$$

where $\delta(t)$ is a shorthand for $\delta(\pi(t))$.

If t belongs to the set of initial events of the unfolding (${}^\bullet t = \emptyset$), then the probability that the firing time is equal to X it the probability that its delay is X. Otherwise, the probability that $f(t)$ is X is the probability that the chosen delay for t plus the maximum of the firing times of the input transitions (considering their probability of being chosen if the transition is early-evaluated) is X.

Theorem 1 (Steady state throughput using firing times) *Let G be a TGV, and t be a transition of G. The steady state throughput of t is:*

$$\Theta(t) = \lim_{i\to\infty} \frac{i}{\overline{f}(t_i)} \tag{6.4}$$

where t_i is the i-th instantiation of t in the unfolding of G, and $\overline{f}(t_i)$ is the average of the firing time for transition t_i.

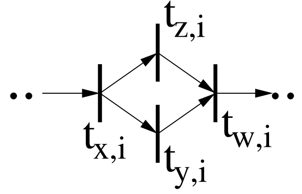


Figure 6.4: Portion of an unfolding with reconvergent paths.

Proof. By definition 3, the throughput is $\sigma(t,\tau)/\tau$, where the time τ tends to infinity. Since t_i is the i-th instantiation of t in the unfolding of G, t will have fired i times at time $\overline{f}(t_i)$, on average. Therefore, if $\tau = \overline{f}(t_i)$, then

$$\frac{\sigma(t,\tau)}{\tau} = \frac{i}{\overline{f}(t_i)}$$

□

6.3 Max-plus Algebra with Early Evaluation

Instead of directly computing the firing time of each event in the unfolding, a symbolic expression for each firing time is created first. The advantage of using symbolic expressions is that they can capture correlations that are lost otherwise. Consider the portion of an unfolding shown in Fig. 6.4. Assume that the firing time of $t_{x,i}$ is $f(t_{x,i}) = [(4, 0.2), (5, 0.8)]$. Then, $f(t_{y,i}) = f(t_{x,i}) + \delta(t_{y,i})$ and $f(t_{z,i}) = f(t_{x,i}) + \delta(t_{z,i})$. If both $t_{y,i}$ and $t_{z,i}$ have unit delay, then $f(t_{y,i}) = [(5, 0.2), (6, 0.8)]$ and $f(t_{z,i}) = [(5, 0.2), (6, 0.8)]$. To compute the arrival time of $t_{w,i}$, one must compute the maximum of $f(t_{y,i})$ and $f(t_{z,i})$. The resulting firing time if each possible combination is taken into account is:

$$\max\left(\begin{bmatrix} (5, 0.2) \\ (6, 0.8) \end{bmatrix}, \begin{bmatrix} (5, 0.2) \\ (6, 0.8) \end{bmatrix}\right) = \begin{bmatrix} (\max(5,5), 0.04) \\ (\max(5,6), 0.16) \\ (\max(6,5), 0.16) \\ (\max(6,6), 0.64) \end{bmatrix} = \begin{bmatrix} (5, 0.04) \\ (6, 0.96) \end{bmatrix}$$

The previous computation gives a too conservative upper bound. Firing times $f(t_{y,i})$ and $f(t_{z,i})$ are correlated, because they both depend on $f(t_{x,i})$. It is never the case that $f(t_{y,i}) \neq f(t_{z,i})$. The correct computation is $\max(f(t_{y,i}), f(t_{z,i})) = [(5, 0.2), (6, 0.8)]$. Using symbolic expressions, some

correlations can be structurally detected and fixed. For example, if $\delta(t_{w,i}) = 0$, $f(t_{w,i})$ from Fig. 6.4 is:

$$
\begin{aligned}
f(t_{w,i}) &= 0 + \max(f(t_{x,i}) + \delta(t_{y,i}), f(t_{x,i}) + \delta(t_{z,i})) \\
&= f(t_{x,i}) + \max(\delta(t_{y,i}), \delta(t_{z,i})) \\
&= [(4, 0.2), (5, 0.8)] + \max(1, 1) \\
&= [(5, 0.2), (6, 0.8)]
\end{aligned}
$$

This section defines an extension to max-plus algebra that allows modeling variable delays and early evaluation. It is shown how the algebra expressions can be build from an unfolding and how to evaluate them. Two algorithms are provided:

- **A_exact,** defined in Section 6.3.4, computes the exact probability distribution of each firing time, but it has an exponential worst-case complexity.
- **A_prune,** defined in Section 6.3.5, is a variation of A_exact which allows to efficiently obtain a *statistical upper bound* of each firing time. Since A_prune returns *upper bounds* of the actual firing times $\overline{f}(t)$, the throughput of the system obtained from this method will be a *lower bound* of the actual throughput (see equation 6.4).

6.3.1 Definitions

Max-plus algebra [5] has been previously used in the literature in order to find the cycle time of asynchronous circuits with AND-causality. It has also been extended to Min-Max functions in order to perform timing analysis of asynchronous circuits with OR-causality and latch-controlled synchronous circuits with clock schedules [70].

In these papers, AND-causality join structures are translated to max functions and OR-causality to min functions. In order to adapt max-plus algebra to elastic systems with early evaluation and variable latency, besides max operation, a new early-evaluation operation is needed. Furthermore, leaf variables are sets of delay-probability pairs instead of positive real numbers. While some properties of Max-Min functions are probably lost, some basic useful properties are still maintained in this algebra.

Definition 5 (MPEE expression) *A max-plus expression with early evaluation (MPEE), e, is a term in the grammar:*

$$e \coloneqq \ \delta \mid x \mid e_1 + e_2 \mid e_1 \wedge e_2 \mid e_1^{\alpha_1} \vee e_2^{\alpha_2}$$

where δ is a constant, x is a variable, e_1 and e_2 are MPEE expressions, and α_i are early-evaluation probabilities. Constants are sets of delay-probability pairs, and variables must be assigned to sets of delay-probability pairs for evaluation. For every early-evaluation operator, the sum of probabilities must be 1.

Operator $+, \wedge, \vee$ represent the sum, max and early-evaluation operations for sets of delay-probability. Let us define a grouping operation on sets of pairs by a simple example:

$$group\{(1, 0.1), (2, 0.3), (1, 0.6)\} = \{(1, 0.7), (2, 0.3)\}$$

Notice that grouping does not change the average delay or any other statistical metric of the set of delays. Now the operations on delay-probability pairs can be defined as follows:

Definition 6 (Addition of two sets of delay-probability pairs) *Given two sets of delay-probability pairs, $\delta_1 = \{(d_1, p_1)\}$, $\delta_2 = \{(d_2, p_2)\}$, their addition is the set of delay-probability pairs*

$$group\{(d_1 + d_2, p_1 \times p_2) \mid (d_1, p_1) \in \delta_1 \wedge (d_2, p_2) \in \delta_2\}$$

For example:

$$\begin{bmatrix} (2, 0.4) \\ (3, 0.6) \end{bmatrix} + \begin{bmatrix} (1, 0.8) \\ (2, 0.2) \end{bmatrix} = group\left(\begin{bmatrix} (3, 0.32) \\ (4, 0.08) \\ (4, 0.48) \\ (5, 0.12) \end{bmatrix}\right) = \begin{bmatrix} (3, 0.32) \\ (4, 0.56) \\ (5, 0.12) \end{bmatrix}$$

Definition 7 (Maximum of two sets of delay-probability pairs) *Given two sets of delay-probability pairs, $\delta_1 = \{(d_1, p_1)\}$, $\delta_2 = \{(d_2, p_2)\}$, their maximum is the set of delay-probability pairs*

$$group\{(\max(d_1, d_2), p_1 \times p_2) \mid (d_1, p_1) \in \delta_1 \wedge (d_2, p_2) \in \delta_2\}$$

For example:

$$\begin{bmatrix} (2,0.4) \\ (3,0.6) \end{bmatrix} \wedge \begin{bmatrix} (1,0.8) \\ (4,0.2) \end{bmatrix} = group\left(\begin{bmatrix} (2,0.32) \\ (4,0.08) \\ (3,0.48) \\ (4,0.12) \end{bmatrix}\right) = \begin{bmatrix} (2,0.32) \\ (3,0.48) \\ (4,0.20) \end{bmatrix}$$

Definition 8 (Early-evaluation operation) *Given two sets of delay-probability pairs,* $\delta_1 = \{(d_1, p_1)\}$*,* $\delta_2 = \{(d_2, p_2)\}$*, and two real numbers,* α_1, α_2 *such that* $0 \le \alpha_1, \alpha_2 \le 1$ *and* $\alpha_1 + \alpha_2 = 1$*, the early-evaluation operation results in the set of delay-probability pairs*

$$group\{\bigcup_{i\in\{1,2\}} \{(d_i, p_i \times \alpha_i) \mid (d_i, p_i) \in \delta_i\}\}$$

For example:

$$\begin{bmatrix} (2,0.4) \\ (3,0.6) \end{bmatrix}^{0.1} \vee \begin{bmatrix} (1,0.8) \\ (4,0.2) \end{bmatrix}^{0.9} = \begin{bmatrix} (2,0.04) \\ (3,0.06) \end{bmatrix} \bigcup \begin{bmatrix} (1,0.72) \\ (4,0.18) \end{bmatrix} = \begin{bmatrix} (1,0.72) \\ (2,0.04) \\ (3,0.06) \\ (4,0.18) \end{bmatrix}$$

It is assumed that + has a higher binding than $\wedge$ or $\vee$. All operations are associative and commutative. Hence, although they have arity two, they can be easily extended to arity n. Addition distributes over both $\wedge$ and $\vee$:

$$\begin{aligned} e_1 + (e_2 \wedge e_3) &= e_1 + e_2 \wedge e_1 + e_3 \\ e_1 + (e_2^{\alpha_1} \vee e_3^{\alpha_2}) &= (e_1 + e_2)^{\alpha_1} \vee (e_1 + e_3)^{\alpha_2} \end{aligned} \tag{6.5}$$

6.3.2 Representation of MPEE Expressions

Given an unfolding of a TGV, an MPEE expression is built for each transition. An MPEE expression is represented as a tree, where each node in the tree is assigned an operator (plus, max or early evaluation). The leaves of the tree are either constants (a set of pairs of delay-probability) or a variable.

Algorithm BuildMPEE presents a method to derive an MPEE expression for a given transition

of an unfolding. It also builds an MPEE expression for each of the input transitions recursively until the beginning of the unfolding is reached.

All MPEE expressions are stored in a library of built expressions. To do so, expressions are stored in a unique table in memory, using a technique similar to BDD databases [22]. This way, memory usage is lower and evaluation of expressions is faster, since results can be reused. This is useful, for example, when several transitions share the same set of inputs, since the expression built to compute the arrival time will be shared. Notice that it will cause the expression to form an acyclic graph instead of a tree.

Algorithm 5: BuildMPEE, construct an MPEE expression from an unfolding.

Input: G : unfolding of a TGV, t : transition of G
Output: An MPEE expression for the firing time of t

if $t \in$ *BuiltExpressions* **then**
    #If the expression has already been created, return it
    return *BuiltExpressions*$[t]$
arrival = $\emptyset$ *#Set of MPEE expressions from the input transitions of t*
for $p \in {}^{\bullet}t$ **do**
    e = BuildMPEE(${}^{\bullet}p$)
    arrival = arrival $\cup \{e\}$
if *Early*(t) **then**
    e_a = MakeEarly(arrival, Probs(t))
else
    arrival = SimplifyMax(arrival)
    e_a = MakeMax(arrival)
e_a = FactorOut(e_a)
e = MakePlus(e_a, MakeLeaf($\delta(t)$))
BuiltExpressions$[t]$ = e
return e

Algorithm BuildMPEE calls the following procedures:

- *Early*. Checks whether the given transition is early evaluation.
- *Probs*. Returns the guard probabilities for the input transitions.
- *MakeLeaf*. Returns an expression associated with the delay of a transition. If it is not a variable-delay transition, then a constant is returned. Otherwise, a new variable is generated.

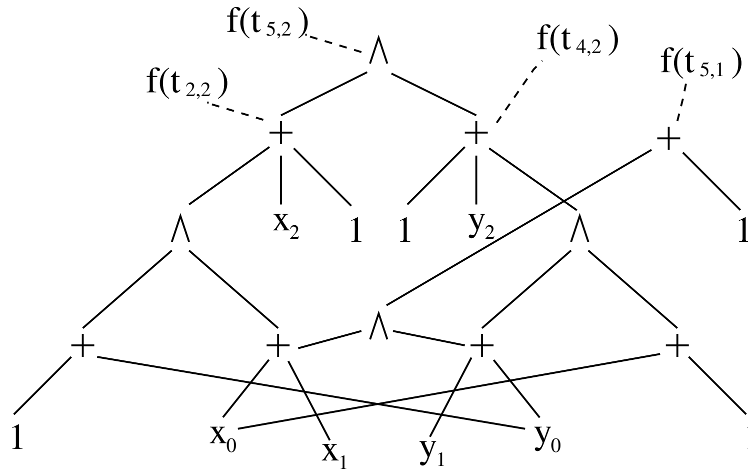


Figure 6.5: Graph representation of an MPEE expression for unfolding in Fig. 6.3(b). x_k represents $\delta(t_{2,k})$ and y_k represents $\delta(t_{4,k})$.

- *MakePlus, MakeMax, MakeEarly*. Given a list of input expressions, a new expression is created which is either the addition, max or early evaluation of the given list. If the list of input expressions has only one element, the same element is returned. MakeEarly receives a list of probabilities. If an expression with the same operators and operation already exists in memory, then the existing expression is returned. A lower and upper bound for the expression is computed and recorded. Some basic simplifications are done, e.g., constants are grouped ($0 \wedge 1 \wedge e_0 = 1 \wedge e_0$).

- *SimplifyMax*. Given a list of expressions, it checks whether some of them is subsumed by the rest when computing the maximum of the expressions. It uses the upper and lower bounds of the expressions.

- *FactorOut*. Given an early or max expression, it applies the distributive property in order to factor out common sub-expressions.

Given the following delays $\delta(t_0) = \delta(t_5) = 0, \delta(t_1) = \delta(t_3) = 1, \delta(t_2) = \delta(t_4) = [(1, 0.5), (2, 0.5)]$; the firing time of $t_{5,2}$ ($f(t_{5,2})$) in Fig. 6.3(b) is described by the expression graph in Fig. 6.5. In this figure, x_k is a short-cut for $\delta(t_{2,k})$ and y_k is a short-cut for $\delta(t_{4,k})$. It can be seen how expressions are reused to compute different firing times, such as $f(t_{5,2})$ or $f(t_{5,1})$.

6.3.3 Lower and Upper Bounds of Expressions

As it will be shown later, every time a max expression is created, new reconvergent paths may be created. Reconvergent paths are directly correlated to the time needed to accurately evaluate an expression. Therefore, each time a max expression is created, it is useful to try to discard some of the sub-expressions, in order to increase the efficiency of the exact method and the accuracy of the lower bound method that will be presented in Section 6.3.5. To this purpose, the lower and upper bound of each expression is recorded. These bounds can be computed using the following rules:

Expression	Upper bound	Lower bound
$e = \delta(t)$	$ub(e) = ub(\delta(t))$	$lb(e) = lb(\delta(t))$
$e = e_1 + e_2$	$ub(e) = ub(e_1) + ub(e_2)$	$lb(e) = lb(e_1) + lb(e_2)$
$e = e_1 \wedge e_2$	$ub(e) = \max(ub(e_1), ub(e_2))$	$lb(e) = \max(lb(e_1), lb(e_2))$
$e = e_1 \vee e_2$	$ub(e) = \max(ub(e_1), ub(e_2))$	$lb(e) = \min(lb(e_1), lb(e_2))$

For leaf expressions, the upper and lower bounds are the upper and lower bounds of the corresponding set of delay-probability pairs. If the expression is a sum of the expressions, then the upper bound is the sum of upper bounds, and the lower bound the sum of lower bounds. The same rationale can be applied to the upper and lower bound of a max operation. For an early-evaluation operation, the upper bound corresponds to the case where the selected branch is the one with the greatest upper bound. Similarly, the lower bound corresponds to the case where the selected branch is the one with the lowest lower bound.

The SimplifyMax operation discards redundant expressions given a list of expressions whose maximum is going to be computed. In order to do so, the maximum of the lower bounds is computed (max_lb). Then, for each expression with bounds (lb, ub), if $ub < max_lb$, the expression is discarded.

6.3.4 Exact Evaluation of MPEE Expressions

Once an MPEE expression has been built for every transition, the expressions must be evaluated in order to be able to obtain the throughput. If the evaluation is performed directly applying the operations in definitions 6, 7 and 8, the accuracy of the result decreases for every max operation in which the operands are correlated. In this case, the result does not correspond any more to

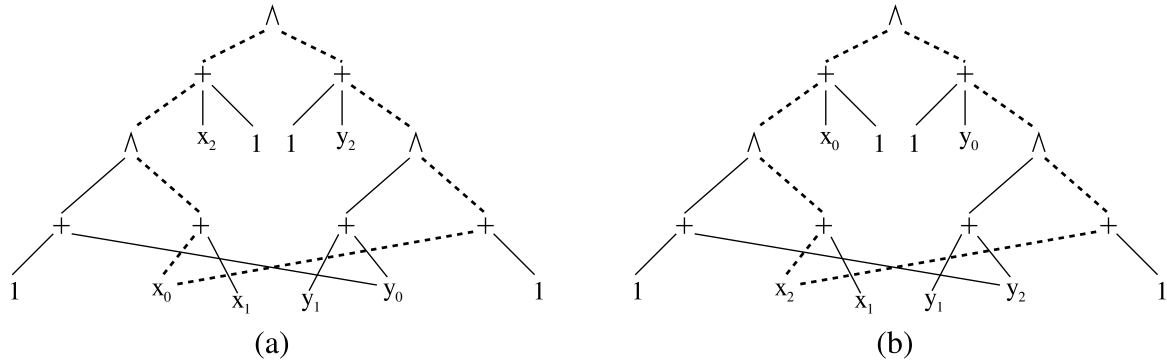


Figure 6.6: (a) MPEE expression where x_0 and y_0 are correlated expressions, the reconvergent paths for x_0 are dashed, (b) equivalent MPEE expression where x_2 and y_2 are correlated expressions.

evaluating equation 6.3 correctly.

The correlations among different operands of a max expression appear due to reconvergent paths, as in Fig. 6.4. Both [2] and [101] identify this problem when doing statistical timing analysis of circuits and propose exact solutions with exponential time cost. They identify nodes that are the sources of reconvergent paths, and then they obtain an exact result by evaluating the resulting delay for each possible combination at the sources of reconvergent paths, merging back all the results to obtain the final result. Given an MPEE expression, it is possible to identify which sub-expressions are the sources of reconvergent paths by traversing the expression.

Definition 9 (Correlated Expression) *Given n MPEE expression $e = e_1 \wedge e_2 \wedge \dots e_n$, an expression e' is correlated with respect to e if there are at least two sub-expressions of e, e_i, e_j, such that there is a path in the expression graph from e_i to e' and from e_j to e'.*

Correlated expressions are the sources of a reconvergent path in the expression graph, which means there is a reconvergent path in the unfolding that could not be factored out. The set of correlated expressions for a given max expression can be found by traversing its expression graph. Correlated expressions which evaluate into a single possible delay are not a problem, because it is not necessary to remember there are combinations of delays which are not possible when the reconvergent paths meet again. The set of troublesome correlated expressions are the ones that can increase the size of a set of delay-probability pairs, which are leaf expressions with more than one possible delay and early-evaluation expressions.

For example, x_0 is a correlated expression in Fig. 6.6(a) with respect to the top max operation of the expression graph. The two possible paths to reach x_0 from the top of the expression are drawn in dotted lines. The expression graph is symmetric, y_0 is also a correlated expression. Symbolic

MPEE expressions can alleviate the problem of reconvergent paths since expression manipulation can factor out common expressions in a max operation. However, there are some cases where this is not possible. For example, in Fig. 6.6(a), x_0 and y_0 can be factored out so that they are not correlated expressions any more, but then x_2 and y_2 become correlated expressions, as shown in Fig. 6.6(b). There is no way to factor out all of them. The steps performed to reach Fig. 6.6(b) from Fig. 6.6(a) are the following ones:

initial expression	$(1 + x_2 + ((1 + y_0) \wedge (x_0 + x_1)))$ $\wedge(1 + y_2 + ((1 + x_0) \wedge (y_0 + y_1)))$
after pushing $(1 + x_2)$ and $(1 + y_2)$ inside	$(2 + x_2 + y_0) \wedge (1 + x_2 + x_1 + x_0)$ $\wedge(2 + y_2 + x_0) \wedge (1 + y_2 + y_1 + y_0)$
after factoring out $(1 + x_0)$ and $(1 + y_0)$	$(1 + x_0 + ((1 + y_2) \wedge (x_2 + x_1)))$ $\wedge(1 + y_0 + ((1 + x_2) \wedge (y_2 + y_1)))$

To obtain the exact result of an expression e even if there are correlated expressions, a case by case evaluation must be performed. For each possible selection of delays in the set correlated expressions, e must be evaluated and its result scaled to the selection probability. This way, whenever a max is evaluated, all combinations of delays among the operands are possible and a regular max evaluation can be performed without loosing any accuracy due to correlations. Algorithm EvaluateRecursive shows how evaluation case by case can be performed using a recursive algorithm.

For each correlated expression, it selects all possibilities one by one and calls the algorithm recursively. The accumulated probability is kept. Once all the correlated expressions have been traversed, the procedure *Eval* is called, which evaluates the expression using definitions 6, 7 and 8. *Eval* is a function that will evaluate sub-expressions if it is needed. After evaluation, the partial result is stored after scaling it with the accumulated probability *prob*.

For the example in Fig. 6.6(a), the correlated expressions are x_0 and y_0. If all variables x_i and y_i have a delay equal to $[(1, 0.5), (2, 0.5)]$, algorithm EvaluateRecursive will branch twice, once for x_0 and once for y_0. Since each variable has two possible delays, there will be 4 calls to *Eval* with the four different combinations of delays:

Algorithm 6: EvaluateRecursive, algorithm that evaluates an MPEE expression.

Input: e : MPEE expression, $i : \mathbb{N}$, CorrelatedExpr : vector of MPEE expressions, $prob : \mathbb{R}$
Output: Set of delay-probability pairs after evaluating expression e scaled to probability *prob* considering only the i-th first correlated expressions in CorrelatedExpr

```
if i = 0 then
    dp = Eval(e) #Evaluate the expression using the selected guards and delays
    for each (d, p) ∈ dp do
        #Scale the probability of each delay by multiplying it by prob
        p = p × prob
    return dp
else
    dp = EmptySetDelayProbability()
    ce = CorrelatedExpr[i]
    if Early(ce) then
        for each guard (g, p) of ce do
            SelectGuard(ce, g)
            dp = group{dp, EvaluateRecursive(e, i − 1, CorrelatedExpr, prob × p)}
    else
        #ce is a leaf expression with variable-delay
        for each delay (d, p) of ce do
            SelectDelay(ce, d)
            dp = group{dp, EvaluateRecursive(e, i − 1, CorrelatedExpr, prob × p)}
    return dp
```

Delay combinations	Result
$x_0 = 1, y_0 = 1$	$[(4, 0.0625), (5, 0.5), (6, 0.4375)]$
$x_0 = 1, y_0 = 2$	$[(5, 0.125), (6, 0.625), (7, 0.25)]$
$x_0 = 2, y_0 = 1$	$[(5, 0.125), (6, 0.625), (7, 0.25)]$
$x_0 = 2, y_0 = 2$	$[(5, 0.0625), (6, 0.5), (7, 0.4375)]$

Each combination has a probability of 0.25. The final probability distribution is obtained by scaling each distribution by its probability and grouping them together:

$$[(4, 0.015625), (5, 0.203125), (6, 0.546875), (7, 0.234375)].$$

The final average firing time is 6.0.

Notice that when an expression is evaluated, all its sub-expressions must also be evaluated.

These sub-expressions correspond to the input cone of the transition in the unfolding. Thus, it is only necessary to evaluate the expressions of the newest transitions in the unfolding to obtain all the firing times.

Algorithm 7: A_exact, algorithm that evaluates an MPEE expression.

Input: e : MPEE expression
Output: Set of delay-probability pairs after evaluating expression e

CorrelatedExpr = FindCorrelatedExpressions(e)
numberCorrelatedExpr = *size*(CorrelatedExpr)
#Call the recursive algorithm that will actually evaluate the expression
EvaluateRecursive(e, numberCorrelatedExpr), CorrelatedExpr, 1.0)

Algorithm A_exact evaluates a symbolic expression and all its sub-expressions. It takes into account all correlations, and thus, it computes the exact probability distribution for the firing time of each transition in the unfolding. It finds the set of correlated expressions structurally by traversing the expression, and then it calls EvaluateRecursive. Assuming the length of the unfolding is large enough, the throughput can be obtained by using equation 6.4.

6.3.5 Upper Bound Evaluation of MPEE Expressions

For most examples, A_exact is infeasible because it is not possible to factor out all correlations in the MPEE expressions, and the run-time becomes exponential. Algorithm A_prune introduces a couple of heuristics that significantly reduce its complexity. First, the size of the sets of delay-probability pairs must be kept small enough so that operations do not take too much time. Second, some correlated variables and early-evaluation expressions can be ignored and computed as if they did not add reconvergent paths. A_prune computes an upper bound of the firing times of each transition. Therefore, the obtained throughput will be a lower bound of the actual throughput.

Reduction of the Size of Sets of Delay-Probability Pairs

As the expressions grow and have a larger depth, the number of possible delays in the solution also grows. Computing max and plus operations has a quadratic cost on the size of the operands. Thus, if $\delta_0 \wedge \delta_1$ is computed, and δ_0 is a set of delay-probability pairs with 20 possible delays, and δ_1 has another 20 possible delays, 400 max operations plus 400 multiplications of probabilities will be computed.

Corner cases tend to have smaller and smaller probabilities. If a variable-delay transition t has delay 1 with probability of 0.1, after 10 periods of the unfolding, there will be one delay-probability pair with probability 10^{-10} corresponding to the case where all instantiations of t selected delay 1.

The probability distribution of the firing time of transitions after some periods of the unfolding typically has the form of a mountain chain, where the greatest delays and the smallest delays have the smallest probabilities. One optimization that can reduce the running time with no significant cost on the accuracy of the results is to remove delays with low probability.

After each operation, the set of delays is traversed following increasing order. Neighboring expressions with probabilities lower than a threshold are grouped together into the next delay which has a probability higher than the threshold, forming a new pair whose delay is the weighted mean of the delays being grouped, to ensure that the total average delay does not change. For example, consider the delay $[\ldots, (9, 0.2), (10, 10^{-4}), (11, 1.5 \times 10^{-4}), (12, 0.1), (13, 0.2), \ldots]$. If the threshold is 10^{-3}, then delay 9 remains unchanged, and delays 10 and 11 are merged into delay delay 12, which has a probability larger than 10^{-3}. The resulting delay d and probability p are computed as follows:

$$p = 10^{-4} + 1.5 \times 10^{-4} + 0.1 = 0.10025$$
$$d = (10 \times 10^{-4} + 11 \times 1.5 \times 10^{-4} + 12 \times 0.1)/p = 11.9965$$

Therefore, the new delay is $[\ldots, (9, 0.2), (11.9965, 0.10025), (13, 0.2), \ldots]$.

A high threshold may force too much grouping, which may actually slow down the computation. Empirically, this value has been set to 10^{-20}. To ensure that results do not grow too much, if the size of the set is larger than some value, for example, 64, some extra grouping may be performed with a larger threshold. Notice that this optimization has a negligible impact on the final average firing times, since the probability threshold has been set to a value that is small enough.

Ignoring Correlated Expressions

Some correlations may be completely ignored. In order to do so, the initial call to Algorithm EvaluateRecursive can be changed so that it does not traverse the whole list of correlated expressions, as shown in Algorithm EvaluateUB.

If $max_correlated_expr = 2$, then only the two first correlated expressions in CorrelatedExpr will be considered. Algorithm A_prune ignores all correlations that have not been simplified during the creation of the expressions. It is equivalent to EvaluateUB when $max_correlated_expr = 0$. Notice that even if $max_correlated_expr$ is 0, some correlations are still taken into account,

Algorithm 8: EvaluateUB, algorithm that computes an upper bound of the result of evaluating an MPEE expression.

Input: e : MPEE expression, $max_correlated_expr$: $\mathbb{N}$
Output: Set of delay-probability pairs after evaluating expression e considering only $max_correlated_expr$ correlated expressions.

CorrelatedExpr = FindCorrelatedExpressions(e)
numberCorrelatedExpr = min($max_correlated_expr$, $size$(CorrelatedExpr))
EvaluateRecursive(e, numberCorrelatedExpr), CorrelatedExpr, 1.0)

Algorithm 9: A_prune, algorithm that computes an upper bound of all firing times by ignoring all correlations during evaluation.

Input: e : MPEE expression
Output: Set of delay-probability pairs after evaluating expression e considering no correlations during evaluation.

EvaluateRecursive($e, 0, \emptyset$, 1.0)

since they are captured and simplified away when building the MPEE expression in Algorithm BuildMPEE.

For the example in Fig. 6.5, if only correlations due to x_0 are taken into account, only two evaluations must be done:

Delay combinations	Result
$x_0 = 1$	$[(4, 0.015625), (5, 0.296875), (6, 0.5625), (7, 0.125)]$
$x_0 = 2$	$[(5, 0.09375), (6, 0.5625), (7, 0.34375)]$

Since each combination has a probability of 0.5, the final result is

$$[(4, 0.0078125), (5, 0.1953125), (6, 0.5625), (7, 0.234375)],$$

which has an average of 6.02. The exact average firing time was 6. If no correlations are taken into account, then the final result is

$$[(4, 0.003906), (5, 0.1875), (6, 0.574219), (7, 0.234375)],$$

which has an average of 6.04. In general, the average firing time increases as the number of correlations considered decreases.

Upper Bound Theorem

As the previous example shows, ignoring correlations yields an upper bound of the firing time. Since the throughput of an elastic system is computed using $i/f(t_i)$, the throughput computed ignoring correlated expressions will be a *lower bound* of the real throughput.

Definition 10 (Definition 5 from [2]) *A cumulative distribution function (CDF) $Q(x)$ is a statistical upper bound of another CDF $S(x)$ if and only if for all x, $Q(x) \leq S(x)$.*

Using definition 10, it can be stated that, for each transition t_i of the unfolding, the firing time probability distribution that A_prune computes for t_i is a statistical upper bound of the actual firing time probability distribution of t_i.

Theorem 2 (Theorem 2 in [2]) *Let Q, R, S be independent random variables with a lower bound and an upper bound. Let Q_1, Q_2 be random variables with CDFs that are identical to the CDF of Q. The random variable $Q_1 + R \wedge Q_2 + S$ is an upper bound of the random variable $Q + R \wedge Q + S$.*

Let us illustrate this theorem using a simple example. Consider $Q = Q_1 = Q_2 = [(1, 0.3), (3, 0.7)]$, $R = 2, S = 1$. Then, $Q_1 + R = Q + R = [(3, 0.3), (5, 0.7)]$ and $Q_2 + S = Q + S = [(2, 0.3), (4, 0.7)]$. Thus, $Q + R \wedge Q + S = Q + R = [(3, 0.3), (5, 0.7)]$ since we consider only Q. However, since Q_1 and Q_2 are independent, $Q_1 + R \wedge Q_2 + S = [(3, 0.09), (4, 0.21), (5, 0.7)]$. The case in which correlations are ignored has one more possible delay (4), and it is a statistical upper bound of the case in which correlations are considered.

This theorem can be extended to handle early-evaluation MPEE expressions using the following lemmas.

Lemma 3 *Let Q be a random variable with a lower bound and an upper bound. Let Q_1, Q_2 be independent random variables with CDFs that are identical to the CDF of Q. The CDF of the random variable $Q_1^{\alpha_1} \vee Q_2^{\alpha_2}$ is equal to the CDF of the random variable $Q^{\alpha_1} \vee Q^{\alpha_2}$.*

Proof. Using the definition of the early-evaluation operation on sets of delay-probability pairs, $Q^{\alpha_1} \vee Q^{\alpha_2} = \{(d, p \times \alpha_1) \mid (d, p) \in Q\} \bigcup \{(d, p \times \alpha_2) \mid (d, p) \in Q\}$, and $Q_1^{\alpha_1} \vee Q_2^{\alpha_2} = \{(d, p \times \alpha_1) \mid (d, p) \in Q_1\} \bigcup \{(d, p \times \alpha_2) \mid (d, p) \in Q_2\}$. Since the pairs of Q_1 and the pairs of Q_2 are not combined in any way, a bijection can be built between delay-probability pairs from $Q^{\alpha_1} \vee Q^{\alpha_2}$ and delay-probability pairs from $Q_1^{\alpha_1} \vee Q_2^{\alpha_2}$. Therefore, they have the same CDF. □

Lemma 4 *Let R, S be independent random variables with a lower bound and an upper bound. Let Q be the random variable obtained by $R^{\alpha_1} \vee S^{\alpha_2}$. Let R_1, R_2 be independent random variables with the same CDF as R and S_1, S_2 be independent random variables with the same CDF as S. Finally, let Q_1, Q_2 be independent random variables built by the expression $Q_i = R_i^{\alpha_1} \vee S_i^{\alpha_2}, i \in \{1, 2\}$. The random variable $Q_1 \wedge Q_2$ is an upper bound of the random variable $Q \wedge Q$.*

Proof. It must be shown that for every possible x, $\mathbb{P}(Q_1 \wedge Q_2 \leq x) \leq \mathbb{P}(Q \wedge Q \leq x)$.

Using the maximum and early evaluation definition:

$$\begin{aligned}
\mathbb{P}(Q \wedge Q \leq x) = & \ \alpha_1 \times \mathbb{P}(R \wedge R \leq x) + \alpha_2 \times \mathbb{P}(S \wedge S \leq x), \\
\mathbb{P}(Q_1 \wedge Q_2 \leq x) = & \ \alpha_1^2 \times \mathbb{P}(R_1 \wedge R_2 \leq x) \\
+ & \ \alpha_1 \times \alpha_2 \times \mathbb{P}(R_1 \wedge S_2 \leq x) \\
+ & \ \alpha_2 \times \alpha_1 \times \mathbb{P}(S_1 \wedge R_2 \leq x) \\
+ & \ \alpha_2^2 \times \mathbb{P}(S_1 \wedge S_2 \leq x).
\end{aligned}$$

In the second case, Q_1 can choose one guard and Q_2 can choose the other guard since there is no correlation between them. Next, it will be shown that

$$\alpha_1 \times \mathbb{P}(R \wedge R \leq x) \geq \alpha_1^2 \times \mathbb{P}(R_1 \wedge R_2 \leq x) + \alpha_1 \times \alpha_2 \times \mathbb{P}(R_1 \wedge S_2 \leq x).$$

The same can be shown for the second halves of the previous expressions.

Using the definition of the maximum operation (definition 7),

$$\begin{aligned}
\alpha_1 \times \mathbb{P}(R \wedge R \leq x) = & \ \alpha_1 \times \sum\{p \mid (d, p) \in R \wedge d \leq x\}, \\
\alpha_1^2 \times \mathbb{P}(R_1 \wedge R_2 \leq x) = & \ \alpha_1^2 \times \sum\{p_1 \times p_2 \mid (d_1, p_1) \in R_1 \wedge (d_2, p_2) \in R_2 \wedge \max(d_1, d_2) \leq x\} \\
\alpha_1 \times \alpha_2 \times \mathbb{P}(R_1 \wedge S_2 \leq x) = & \ \alpha_1 \times \alpha_2 \times \sum\{p_1 \times p_2 \mid (d_1, p_1) \in R_1 \wedge (d_2, p_2) \in S_2 \wedge \max(d_1, d_2) \leq x\}.
\end{aligned}$$

Each expression can be partitioned depending on which d of the first operand (R or R_1) it belongs to. For the first one, for each pair $(d, p) \in R$ such that $d \leq x$, its contribution to the total sum is $\alpha_1 \times p$. For the second one, its contribution is $\alpha_1^2 \times p \times \sum\{p_2 \mid (d_2, p_2) \in R_2 \wedge \max(d, d_2) \leq x\}$ and for the third one it is $\alpha_1 \times \alpha_2 \times p \times \sum\{p_2 \mid (d_2, p_2) \in S_2 \wedge \max(d, d_2) \leq x\}$.

Then, $\alpha_1 \times p$ can be removed because it is present in all expressions:

$$\begin{array}{rl} \alpha_1 \times p \geq & \alpha_1^2 \times p \times \sum\{p_2 \mid (d_2, p_2) \in R_2 \wedge \max(d, d_2) \leq x\} \\ + & \alpha_1 \times \alpha_2 \times p \times \sum\{p_2 \mid (d_2, p_2) \in S_2 \wedge \max(d, d_2) \leq x\} \\ \textbf{implies} & \\ 1 \geq & \alpha_1 \sum\{p_2 \mid (d_2, p_2) \in R_2 \wedge \max(d, d_2) \leq x\} \\ + & \alpha_2 \sum\{p_2 \mid (d_2, p_2) \in S_2 \wedge \max(d, d_2) \leq x\} \end{array}$$

Since $\sum\{p_2 \mid (d_2, p_2) \in R_2 \wedge \max(d, d_2) \leq x\} \leq 1$ and $\sum\{p_2 \mid (d_2, p_2) \in S_2 \wedge \max(d, d_2) \leq x\} \leq 1$ by definition (the sum of probabilities of a set of delay-probability pairs is 1), and $\alpha_1 + \alpha_2 = 1$, the previous inequality holds. □

Finally, the theorem that states that MPEE expression evaluation ignoring some correlations provides an statistical upper bound of the evaluation considering all correlations can be stated and proved.

Theorem 5 *Let e be an MPEE expression representing the firing time of transition t_i of the unfolding. Let $Q(x)$ be the CDF of the probability distribution computed by using definitions 6, 7 and 8 on e, and let $S(x)$ be the probability distribution which is the solution of equation 6.3 for transition t_i. $Q(x)$ is a statistical upper bound of $S(x)$.*

Proof. The proof can be partitioned into different parts depending on the structure of e. $S(x)$ corresponds to evaluating e using the exact method which has already been presented.

If e is a leaf, then the solution is the probability distribution given by the $\delta(t)$. Therefore, $Q(x) = S(x)$. Lemma 3 shows that theorem 5 holds when e is an early-evaluation operation, theorem 2 and lemma 4 show that theorem 5 holds when e is a maximum. This theorem does not hold for addition, i.e., $Q_1(x) + Q_2(x)$ is not an statistical upper bound of $Q(x) + Q(x)$, where Q, Q_1 and Q_2 are identical probability distributions. For example, if $Q = Q_1 = Q_2 = [(0, 0.5), (1, 0.5)]$, $Q + Q = [(0, 0.5), (2, 0.5)]$ and $Q_1 + Q_2 = [(0, 0.25), (1, 0.5), (2, 0.25)]$. The mean in both cases is 1, but $[(0, 0.25), (1, 0.5), (2, 0.25)]$ is not an statistical upper bound of $[(0, 0.5), (2, 0.5)]$. However, by construction of MPEE expressions in unfoldings, the same expression is never added to itself, it is always added to the delay of another transition. If the same expression was added to itself, it would mean that an event is being counted twice. Therefore, this case can be ignored. □

6.3.6 Algorithm Complexity

This section determines the time complexity of obtaining the firing times of an unfolding of K periods using algorithms A_exact and A_prune.

The time required to build an unfolding of K periods is $O(m \times K)$, where m is the number of arcs in the original marked graph [118]. Algorithm BuildMPEE creates an MPEE expression for each transition of the unfolding. It also updates the upper bound and lower bound value for each expression according to the rules defined in Section 6.3.3. When a maximum expression is created, the upper bound and lower bounds are used to decide whether some of the input expressions are redundant. Also, for every maximum and early evaluation, it is tested whether some of the input expressions (up to 2 levels backwards) are common and they can be factored out. Thus, the complexity of the algorithm is $O(m \times K \times \epsilon^2)$, where ϵ is the maximum in-degree for a transition in the unfolding. Notice that this factor is not related to the size of the input and is typically a small number. The expressions are efficiently stored and accessed. There is a unique copy for each expression. Thus, the size of the collection of expressions is linear with respect to the size of the unfolding.

Algorithm A_exact has a worse case exponential complexity. Algorithm EvaluateRecursive calls itself recursively once for each choice of each correlated expression. In the worst case, the number of correlated expressions can be linear with respect to the number of transitions in the unfolding, since each expression can be a correlated expression. Let c be the maximum number of choices for a correlated expression, then the recursive algorithm branches at most c times for every correlated expression, and then the algorithm complexity is $O(c^{m \times K})$.

Algorithm A_prune does not take into account any correlations. Thus, it directly evaluates all the expressions. Each expression is evaluated by using definitions 6, 7 and 8. The time needed for each operation is quadratic with respect to the size of the set of delay-probability pairs. However, as explained in Section 6.3.5, the size of possible delays is kept small, and it can be considered always smaller than a given constant. Thus, each operation takes constant time. Since the number of expressions is linear with respect to the size of the unfolding, the total time needed to evaluate all expressions is also linear: $O(m \times K)$.

6.4 Number of Periods of the Unfolding

The final question that needs to be answered is how many times the TGV should be unfolded, for both A_exact and A_prune algorithms. In [118], an upper bound of the periods of an unfolding is found, so that it can be guaranteed that the exact cycle time is correctly computed. Unfortunately, variable latency and early evaluation make this result invalid, although some other results from the paper still hold. A dynamic algorithm is used to determine the number of periods for the unfolding. This section presents some theoretical background to justify this dynamic algorithm.

6.4.1 Convergence of A_exact

In order to remove the effect of variations in the initial stages of the timing simulation, before the steady state has been reached, the notion of *event-initiated timing simulation* is introduced. When performing an event-initiated timing simulation, the firing times of the transitions that are not reachable from the initiator are assigned a firing time of 0.

Definition 11 (Event-initiated timing simulation) *A timing simulation is initiated by event e if all the events that are not reachable from e in the unfolding are assumed to have a firing time of 0, i.e.,* $f_e(g) = 0$ *iff (*$e = g$ *or* $e \not\Rightarrow g$*), where (*$e \Rightarrow g$*) means that there is a path in the unfolding from e to g.*

The following proposition helps to prove that the average separation between occurrences of the same transition (and hence the throughput) can be computed using the maximum operation.

Proposition 3 from [118] (Triangular Inequality). *For a* t_i*-initiated timing simulation, the occurrence time of a later instantiation of transition t (*$\overline{f}_{t_i}(t_k)$*),* $k > i$*, is larger than or equal to the sum of any combination of occurrence times for instantiations with occurrence indices between i and k and a total occurrence period equal to* $k - i$*:*

$$\forall t, \forall i \geq 0, \forall k > i : \overline{f}_{t_i}(t_k) \geq \max\{\overline{f}_{t_i}(t_j) + \overline{f}_{t_i}(t_{(i+k-j)}) \mid i < j < k\}$$

This proposition holds also for systems with early evaluation and variable-delay transitions. In an unfolding with a finite number of periods, there is a finite number of delay and guard choices. Assume that for each possible choice (that is, for each combination of delays in the variable-delays transitions and guard selections in early-evaluation units), a separate identical unfolding is created.

In each copy, for each early evaluation transition, the places whose guards have not been selected are removed, and each variable-latency transition has chosen only one delay. In the unfolding of Fig. 6.3(b), if t_2 and t_4 have delay $[(1, 0.5), (2, 0.5)]$, the rest of the transitions have a single latency and there is no early evaluation, there are 4 transitions with delay choices: $t_{2,0}, t_{2,1}, t_{4,0}$, and $t_{4,1}$; and each one can choose between two delays, 1 or 2. Thus, there are 2^4 possible combinations. Since each delay has probability 0.5, the probability of each combination is 0.5^4. The number of combinations grows as the number of periods in the unfolding grows. Each copy of the unfolding is a simple elastic system, with neither early evaluation nor variable latency. Hence, proposition 3 holds for each of these unfoldings as proven in [118].

The probabilistic firing time of each transition for the original unfolding can be obtained by adding the firing time of each combination multiplied by the combination probability. In the previous example, it would be necessary to add the firing time of each of the 2^4 combinations multiplied by 0.5^4. Since each of the unfoldings without early evaluation and variable latency honors proposition 3, and the set of all these unfoldings correspond to all the possible behaviors of the original unfolding, proposition 3 also holds for the probabilistic case. It can be inferred by induction that proposition 3 also holds on an infinite unfolding.

Definition 12 (Separation between occurrences) *The separation between two occurrences of the same transition t, t_i and t_j, is defined as:*

$$\Delta_i(t_j) = \frac{\overline{f}_{t_i}(t_j)}{j - i} \tag{6.6}$$

Using the triangular proposition, the throughput can be related to the separation between occurrences for any transition t:

$$\max_{i,j}\{\Delta_i(t_j)\} \leq 1/\Theta \tag{6.7}$$

That is, the estimation of the throughput computed as an inverse of the average separation between occurrences over some length of simulation is an upper bound on the exact throughput.

The limit from the throughput definition (equation 6.4) exists, as pointed out in Section 6.2.4. Equation 6.7 is monotonically improving the bound because of the maximum operation. The longer the unfolding, the closer equation 6.7 gets to the exact throughput. Furthermore, we can derive from the triangular inequality that $\overline{f}_{t_i}(t_k) \geq \overline{f}_{t_i}(t_j) + \overline{f}_{t_i}(t_{i+k-j})$, which points that the separation between occurrences will keep increasing until $\max\{\Delta_i(t_j)\} = 1/\Theta$. Therefore, we can assure

that we can reach any level of accuracy to the throughput estimation by creating a long enough unfolding:

$$\forall \epsilon > 0,\ \exists\ k > 0,\ \max_{\forall k \geq j > i \geq i_0} \{\Delta_i(t_j)\} \geq (1 - \epsilon) \cdot 1/\Theta \quad (6.8)$$

where ϵ is a small number that can be tuned to decide the acceptable error of the algorithm, and i_0 is a constant that avoids taking into account initial perturbations (empirically set to a tenth of the length of the unfolding). This equation justifies the following algorithm. Iteratively, a number of periods is added to the unfolding. Then, the firing times for the new transitions are computed. Let A_exact(k) be $\max_{\forall k \geq j > i \geq i_0} \{\Delta_i(t_j)\}$, the result obtained by running A_exact algorithm on a k-period unfolding. First, A_exact($T/2$) is computed, where T is the number of tokens in the TGV. Next, $T/2$ periods are added at each iteration, until the result stabilizes:

$$\text{A_exact}(i \cdot T/2) + (1 - \epsilon) \geq \text{A_exact}((i + 1) \cdot T/2),$$

Any constant N can be used in place of $T/2$, but $T/2$ scales together with the size and the complexity of the TGV, and it works well based on experiments. Experimentally, convergence is typically reached with less than $3T$ periods in the unfolding.

Instead of performing an event-initiated timing simulation for every transition in the marked graph, it is enough to do so for a cut-set of the transitions. A cut-set of a marked graph is a set of transitions such that each cycle in the graph contains at least one of the transitions. Ideally, a minimum cut-set would optimize the total run-time, but finding this minimum cut-set is a difficult problem by itself. However, the set of transitions such that they have some token at the inputs is a fast way to obtain a cut-set, since all cycles have at least one token.

6.4.2 Convergence of the Upper Bound Method, A_prune

The dynamic method to decide when to stop adding periods to the unfolding can also be applied to A_prune algorithm. For every transition in the unfolding t_i, A_prune computes a firing time probability distribution $f^{ub}(t_i)$ which is a statistical upper bound of $f(t_i)$. There are two important facts to consider::

1. By construction, the upper bound of both distributions is the same, $ub(f^{ub}(t_i)) = ub(f(t_i))$.

2. Since each max expression which has correlated expressions adds some extra overhead to the upper bound, and no other type of operation modifies the accuracy of the result,

$$\overline{f^{ub}}(t_i) = \overline{f}(t_i) + k \implies \overline{f^{ub}}(t_{i+1}) \geq \overline{f}(t_{i+1}) + k$$

Due to the second item, the estimation of the average occurrence distance based on the upper bound, $\overline{f^{ub}}(t_i)$, cannot decrease with iterations, and due to the first item, it is always limited from above. Hence the difference between the actual firing time and its computed upper bound is bounded, and it cannot decrease if more periods are added to the unfolding.

Let Θ^{lb} be the throughput returned by applying A_prune to an infinite unfolding. Considering that $\overline{f^{ub}}(t_i) \geq \overline{f}(t_i)$, it can be derived that $\Theta^{lb} \leq \Theta$. If the previously defined dynamic method is applied to A_prune, it will return a throughput Θ^{lb}/ϵ, where $\epsilon \leq 1$. It must be ensured that $\Theta^{lb}/\epsilon \leq \Theta$, because the objective of this method is to obtain a lower bound of the throughput. Since A_prune is a fast algorithm, the accuracy parameter ϵ in the dynamic algorithm can be set to a small enough value so that the resulting throughput will either be a lower bound of the Θ or it will be equal to Θ for enough significant digits.

6.4.3 Example

Figure 6.7(b) shows the throughput of the TGV in Fig. 6.7(a) with different number of periods in the unfolding and different correlation strategies. In Fig. 6.7(a), the number below each transition represents its delay. There is one early-evaluation transition and one variable-delay transition. The throughput obtained by simulating the elastic system is 0.537. The throughput obtained by the linear programming method in [85] is 0.618.

The two horizontal lines in Fig. 6.7(b) correspond to the simulation throughput and the LP throughput. A_prune(k) is defined similarly to A_exact(k), the result obtained by running A_prune algorithm on a k-period unfolding. The other three lines correspond to A_exact(k), A_prune(k), and a third algorithm where *max_correlated_expr* (presented in Section 6.3.5) has been set to 12. The reader should keep in mind that the figure shows the throughputs, which are the inverse of the average firing times.

It can be seen that A_exact(k) complies with equation 6.7. The computed throughput is always greater than the simulation throughput, and after 5 periods it converges, returning the exact throughput.

A_prune is initially over the throughput, but after 3 periods it crosses the line representing the simulation throughput. As it has been explained, it is possible that this happens if the unfolding is too short. However, it converges at 0.470, which is indeed a lower bound of the throughput.

The plot for *max_correlated_expr* $= 12$ behaves exactly like A_exact while there are less than 12 correlated expressions, since it can capture all correlations and compute all firing times

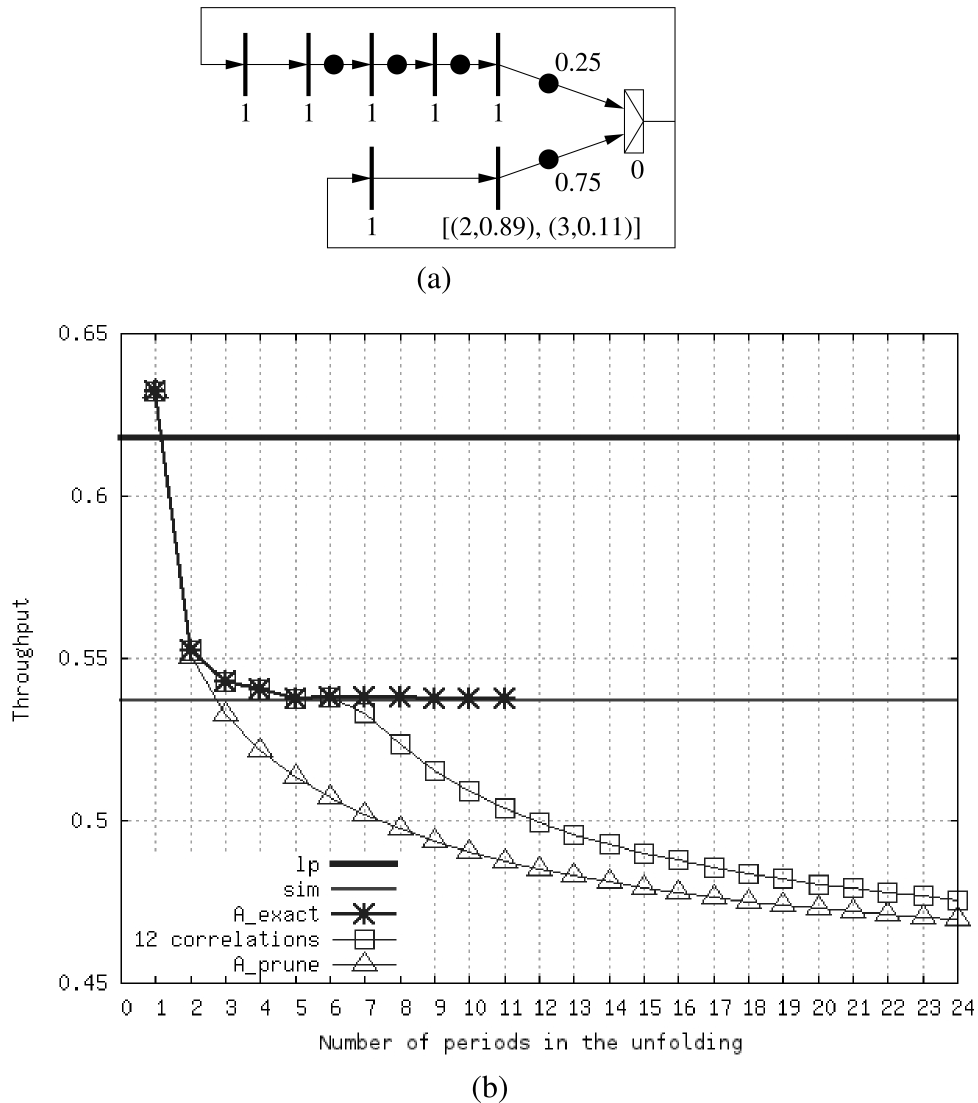


Figure 6.7: (a) TGV with early evaluation and variable-delay (b) Throughput of Fig. 6.7(a) computed using different numbers of unfoldings and different correlation strategies.

correctly. After the sixth period of the unfolding, there are more than 12 correlated expressions in the expression tree, and hence some of the computed firing times are an upper bound of the actual firing times. From this point on, the plot behaves similarly to A_prune. However, it has more accuracy since it takes into account more correlations. The fact that adding more periods to the unfolding reduces the accuracy may seem counter-intuitive. It must be considered that this instance of the algorithm does not compute the throughput, it computes the throughput when

Table 6.1: Experimental results on random graphs with early-evaluation nodes and no variable-latency units. N: nodes, E: edges, EV: early-evaluation nodes, VD: variable-delay nodes, T: tokens, EBs: number of EBs, UB: upper bound throughput (LP), SIM: simulation throughput, LB: lower bound throughput, Δ_{UB}: UB error w.r.t. SIM, Δ_{LB}: LB error w.r.t. SIM.

Graph							Throughput			Error	
Graph	N	E	EV	VD	T	EBs	UB	SIM	LB	Δ_{UB}	Δ_{LB}
G0	9	10	1	0	2	4	0.500	0.500	0.500	0%	0%
G1	9	10	1	0	2	4	0.412	0.412	0.412	0%	0%
G2	9	10	1	0	5	8	0.765	0.732	0.656	5%	-10%
G3	18	28	4	0	10	28	0.619	0.472	0.429	31%	-9%
G4	48	66	6	0	15	39	0.254	0.253	0.253	0%	0%
G5	56	77	6	0	21	46	0.250	0.239	0.235	5%	-2%
G6	54	74	8	0	38	60	0.713	0.702	0.696	2%	-1%
G7	70	94	12	0	33	45	0.867	0.805	0.801	8%	0%
G8	59	142	11	0	53	129	0.250	0.217	0.199	15%	-8%
G9	180	619	47	0	139	340	0.333	0.321	0.305	4%	-5%

only 12 reconvergent paths can be considered, which happens to be a lower bound of the actual throughput. This lower bound is found when the plot converges, at 0.475.

6.5 Experimental Results

This section presents a set of experiments with random graphs. Each graph has been produced in three different flavors, only with early evaluation, only with variable-latency units, and with both early evaluation and variable-latency, to show the accuracy of the presented method in evaluating both features. The applicability of both the lower bound and the upper bound method for performance analysis during a design space exploration is shown on a set of experiments.

6.5.1 Random Graphs

To show the accuracy of the presented method, a set of experiments has been performed on sequential circuits from the MCNC benchmarks. For each circuit, the graph structure was extracted, and the largest strongly connected component was saved. Then, each edge was assigned a token with a random probability between 0.25 and 0.6. Next, each transition was assigned unit delay with a random probability greater than 0.5, otherwise its delay was set to zero. Variable delays were added to 10% of the nodes. These nodes were assigned two random possible delays between

Table 6.2: Experimental results on random graphs with variable-latency units and no early-evaluation nodes. Legend is the same as in Table 6.1.

Graph							Throughput			Error	
Graph	N	E	EV	VD	T	EBs	UB	SIM	LB	Δ_{UB}	Δ_{LB}
G0	9	10	0	1	2	4	0.287	0.287	0.264	0%	-8%
G1	9	10	0	2	2	4	0.107	0.107	0.101	0%	-6%
G2	9	10	0	1	5	8	0.322	0.322	0.308	0%	-4%
G3	18	28	0	2	10	28	0.333	0.199	0.172	67%	-14%
G4	48	66	0	5	15	39	0.161	0.133	0.126	21%	-5%
G5	56	77	0	3	21	46	0.182	0.175	0.172	4%	-2%
G6	54	74	0	7	38	60	0.476	0.265	0.251	80%	-5%
G7	70	94	0	5	33	45	0.526	0.272	0.229	93%	-16%
G8	59	142	0	3	53	129	0.143	0.143	0.143	0%	0%
G9	180	619	0	7	139	340	0.152	0.148	0.127	3%	-14%

0 and 10. Each delay was assigned a random probability. Finally, 25% of the nodes with 2 or more inputs were configured as early-evaluated. The probability of each input was generated randomly.

Tables 6.1, 6.2 and 6.3 show the list of graphs and their throughputs. For each graph, three versions were created. One with only early-evaluation nodes (shown in Table 6.1), one with only variable-delay nodes (shown in Table 6.2), and one with early evaluation and variable-delay (shown in Table 6.3). Results are compared against the upper bound method (LP) in [85] that uses linear programming.

The maximum number of correlations taken into account by the evaluation algorithm was set to 0 in order to obtain the fastest possible run-times. When building the expressions, some correlations can already be factored out. For the case where there is only early evaluation and no variable-delay, the average errors for the upper bound method and the lower bound method are 6.8% and 3.5% respectively. Both methods are quite accurate, although there are two graphs where the error of the upper bound method is over 10%.

The LP method is less accurate on graphs with variable-delay nodes since it does not know the possible delays of these transitions, it only considers the average delay of the transition. The average errors are 26% and 7% respectively. If both early evaluation and variable-delay nodes are taken into account, the average errors are 63% and 9%.

The LP solver used to compute the upper bound method is CPLEX. The presented method has been implemented in C++. The experiments have been run on a Xeon processor at 2.66 GHz with 3 MB of cache. We have found that a simulation of 10000 cycles typically provides an acceptable

Table 6.3: Experimental results on random graphs with early-evaluation nodes and variable-latency units. Legend is the same as in Table 6.1.

Graph							Throughput			Error	
Graph	N	E	EV	VD	T	EBs	UB	SIM	LB	Δ_{UB}	Δ_{LB}
G0	9	10	1	1	2	4	0.372	0.329	0.288	13%	-12%
G1	9	10	1	2	2	4	0.207	0.107	0.101	93%	-6%
G2	9	10	1	1	5	8	0.618	0.538	0.470	15%	-13%
G3	18	28	4	2	10	28	0.474	0.226	0.180	110%	-20%
G4	48	66	6	5	15	39	0.169	0.135	0.126	25%	-7%
G5	56	77	6	3	21	46	0.222	0.181	0.176	23%	-3%
G6	54	74	8	7	38	60	0.692	0.268	0.257	158%	-4%
G7	70	94	12	5	33	45	0.656	0.269	0.236	144%	-12%
G8	59	142	11	3	53	129	0.250	0.199	0.190	26%	-5%
G9	180	619	47	7	139	340	0.233	0.181	0.164	29%	-9%

accuracy, even in graphs which have several variable-latency units with 3 or 4 possible delays and a significant amount of early-evaluation nodes. If there are fewer delay choices in the elastic system, 5000 or even 1000 cycles may be enough simulation time. Figure 6.8 shows the run-times for each of the evaluated methods in the graphs from Table 6.3. The lower bound method is around half to one order of magnitude slower than the LP method. Simulations were around 2 orders of magnitude slower than the lower bound method. The lower bound run for the biggest graph took around 8 seconds. The rest of the runs took less than one second.

6.5.2 Evaluating Relative Performance

If a designer wants to compare several possible designs or perform some architectural exploration, it is important that the order between designs on simulation and on analysis is the same. A set of experiments have been performed in order to determine whether the presented analytical technique can correctly order a set of pipelines. If the throughput of pipeline A is higher that the throughput of pipeline B according to the analytical estimation, then it should also be higher in simulation.

Retiming and recycling [24] has been executed on each graph to extract the set of non-dominated configurations. Each of these configurations is latency equivalent to the original one, and it has been obtained by applying a sequence of elastic transformations, mainly retiming moves and bubble insertion. Then, the throughput of each configuration has been obtained by using the linear programming upper bound method, the presented lower bound method, and simulation. Finally, the configurations have been ordered using each of the throughputs.

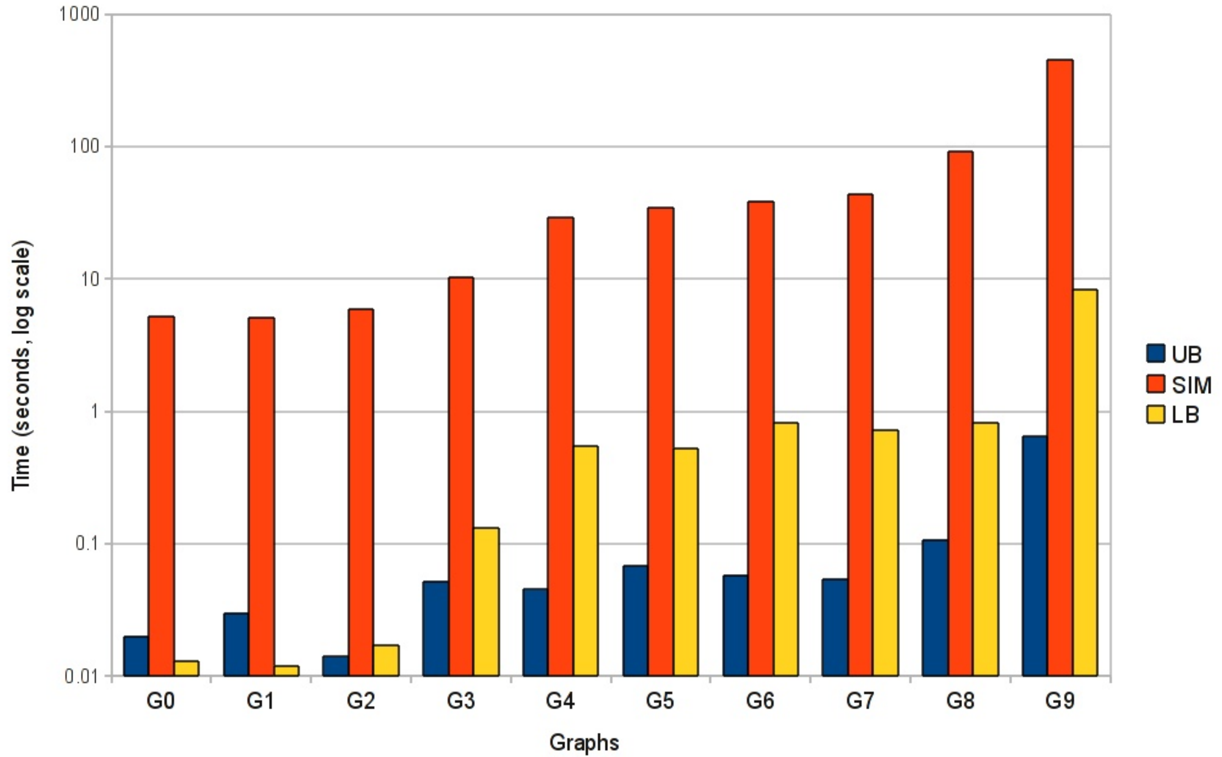


Figure 6.8: Run-time of the different performance evaluation versions in seconds for the graphs in Table 6.3. LB is the presented lower bound method, UB the linear programming upper bound method, and SIM is the simulation time. The Y-axis is in logscale.

Table 6.4 shows the three throughputs (simulation, lower bound and upper bound) for each one of the 16 configurations found by RR for graph G8. The index of each configuration when ordered by the throughput of the corresponding column is written next to each throughput. The reference order is determined by the simulation throughput. In order to achieve the simulation order from the upper bound order, one must do 2 swaps: the 6th and 5th configurations must be swapped, and the 13th and 14th configurations must be swapped. The average throughput difference for each swap is 2.15% (0.8% for the first swap and 3.4% for the second swap). Similarly, 4 swaps are needed to correctly order the order produced by the lower bound analysis, and the average throughput difference per swap is 0.76%. These two metrics can be used to measure the quality of the orderings found by the throughput analysis methods.

Table 6.5 shows the required number of swaps and the average difference between swaps for the upper bound and lower bound methods for each graph in Table 6.3. It is also shown how many

Table 6.4: Configurations found by the retiming and recycling method on graph G8. The first column shows the throughput found by simulation and the order according to this throughput. The second column shows the throughput found by the presented lower bound method and the corresponding order. Finally, the third column shows the throughput according to the upper bound method and the corresponding order.

Sim order	Sim thr	LB order	LB thr	UB order	UB thr
1	0.283	1	0.255	1	0.333
2	0.322	2	0.293	2	0.400
3	0.347	3	0.330	3	0.446
4	0.358	5	0.338	4	0.455
5	0.379	6	0.339	6	0.477
6	0.388	4	0.335	5	0.473
7	0.394	8	0.346	7	0.491
8	0.399	7	0.345	8	0.499
9	0.633	9	0.599	9	0.717
10	0.661	10	0.600	10	0.745
11	0.703	12	0.612	12	0.809
12	0.727	11	0.608	11	0.782
13	0.837	13	0.646	13	0.892
14	0.875	14	0.669	14	0.919
15	0.939	15	0.716	15	0.961
16	1.000	16	1.000	16	1.000

non-dominated configurations were found by retiming and recycling. For most of the graphs, the order found by both the upper bound and lower bound methods is correct. G8 and G9 have more configurations than the rest of the examples and some of the configurations have similar throughputs.

Notice that the throughput difference is small when the ordering found by the analysis methods is not correct. These results validate the method presented in Chapter 5, where at first, analysis does the coarse grain evaluation in order to prune the design space, and then simulation is used in order to find the actual performance of the most promising design points.

6.6 MPEE Simplification using SMT

One interesting optimization that was not integrated to the final solution is to use SAT-module theories (SMT [7, 120]) to further reduce the size of MPEE expressions. Each time a max expression is created, it may add new correlated expressions. Correlated expressions degrade the result of the

Table 6.5: Number of swaps needed to produce a correct order. Confs shows how many RR configurations where created, LB and UB swaps show how many swaps were needed to achieve the correct configuration order, and “swap diff” shows the average throughput difference for the swapped configurations.

Graph	Confs	LB swaps	LB swap diff	UB swaps	UB swap diff
G0	7	0	0.00%	0	0.00%
G1	6	0	0.00%	0	0.00%
G2	3	0	0.00%	0	0.00%
G3	5	0	0.00%	0	0.00%
G4	6	0	0.00%	0	0.00%
G5	4	1	6.44%	0	0.00%
G6	5	0	0.00%	0	0.00%
G7	6	0	0.00%	0	0.00%
G8	16	4	0.76%	2	2.15%
G9	16	1	0.01%	2	6.51%

lower bound method and slow down the exact method. For each operand of the max, SMT may be used to check whether it can become the largest one among the max operands. If it cannot, the operand can be safely discarded. Although this method was implemented and it is theoretically sound, it was not finally used because it was too slow.

This solution can take into account correlations between expressions in order to determine if an expression is redundant (e.g., it can detect that $a+c$ and $b+c$ are redundant in $a+b+c \wedge a+c \wedge b+c$). This is not possible by just looking at the upper and lower bounds of each of the operands of the max expression.

MPEE expressions can be translated to SMT literals using a theory called QF_LIA, i.e., unquantified linear integer arithmetic. This theory is formed by boolean combinations of inequalities between linear polynomials over integer variables. It also allows to model if-then-else expressions. The rules used to translated MPEE expressions into SMT literals are the following ones:

MPEE expression	SMT literal
$e = x, \quad x \in \mathbb{N}$	$e = x$
$e = [(x_0, p_0), (x_1, p_1)], \quad x_i \in \mathbb{N}$	$e =$ **if** c **then** x_0 **else** x_1, where c is a fresh boolean variable.
$e = e_1 + e_2$	$e = e_1 + e_2$
$e = e_1 \wedge e_2$	$e =$ **if** $(e_1 > e_2)$ **then** e_1 **else** e_2
$e = e_1^{\alpha_1} \vee e_2^{\alpha_2}$	$e =$ **if** c **then** e_1 **else** e_2, where c is a fresh boolean variable.

Variable delays are translated to if-then-else expressions where a free boolean variable models that any of the possible delays can be chosen. It can be extended to any number of possible delays, e.g., for three possible delays:

$$e = \textbf{ if } c_0 \textbf{ then } x_0 \textbf{ else } (\textbf{if } c_1 \textbf{ then } x_1 \textbf{ else } x_2)$$

Addition is directly translated, max is translated to if-then-else where the condition checks which is the largest delay, and early evaluation is again modeled by a free boolean variables, meaning that any of the input expressions can be chosen.

When a new max expression is built, it is checked whether some of the operands are redundant. For example, consider a new expression $e = e_1 \wedge e_2 \wedge e_3$. Then, the following problem is given to the SMT solver:

Is $e_1 > e_2$ *and* $e_1 > e_3$ satisfiable?

If the problem is unsatisfiable, it means that it is impossible that e_1 is the greatest of the delays, for any of the possible values of the boolean variables. Therefore, e_1 can be dropped, $e = e_2 \wedge e_3$. If the problem is satisfiable, then e_1 cannot be discarded. The same must be done to decide whether to discard e_2 and e_3. Notice that the order in which the sub-expressions are tested is relevant. If e_1 is already discarded, the problem that will be solved is the satisfiability of $(e_2 > e_3)$. It may be the case that either e_1 and e_2 can be discarded, but not both at the same time.

This optimization was implemented using an incremental SMT solver [60]. This way, it is possible to store a context with the already defined clauses and literals, pushing and popping new clauses and variables. However, it was not efficient to solve a new SMT problem for every operand of every new max expression. It did not scale to large unfoldings in complicated marked graphs.

6.7 Conclusion

This chapter has proposed a method to analyze the performance of elastic systems with early evaluation and variable-latency units. Two methods have been proposed, an exact method with an exponential time cost, and a simplification which can efficiently compute a lower bound.

All the needed features of elastic systems can be modeled using a Timed Multi-Guarded Marked Graph with Variable-Delay (TGV). In order to evaluate the throughput of a TGV, a timing

simulation is performed for a limited number of cycles. First, the TGV is unfolded, and for each of the transitions in the unfolded graph, a max-plus algebra expression with early evaluation (an MPEE expression) is created in order to formulate the firing time of the transition. The advantage of using symbolic expressions is that some of the correlations due to reconvergent paths can be captured and factored out.

Finally, all expressions are evaluated in order to evaluate the throughput of the system. Since early evaluation and variable latency are probabilistic, a probability distribution is obtained when evaluating MPEE expressions. The mean of the distribution is the average firing time of the transition. If the expressions are evaluated taking into account all correlations, the exact firing times, and thus the exact throughput, can be computed. However, exact evaluation has an exponential worst-case run-time. By ignoring the correlations that have not been captured and removed in the MPEE expression, the throughput can be computed in linear time with respect to the size of the unfolding, but the returned value is not exact, it is a lower bound of the actual throughput.

The length of the timing simulation is determined dynamically. Typically, it is enough to use once or twice the number of tokens in the TGV in order to converge with enough confidence on the quality of the result. It has been proven that the required length to compute the throughput always converges.

Results show that it is possible to obtain a tight lower bound of the throughput. The average error of the lower bound method is 6.5%. If this is added to the known method to compute an upper bound of the throughput, it means that the effective cycle time is completely bounded. Applied to the method proposed in the previous chapter, it can help provide more confidence when discarding a configuration: it is possible to safely discard design points whose effective cycle time lower bound is larger than the effective cycle time upper bound of the best design point found so far (unless it provides a good performance/area trade-off).

As a future research direction, it may be possible to develop methods to further simplify the MPEE expressions using the fact that unfoldings have a repetitive structure. For example, SMT could be used to discard late inputs in max expressions during the first periods of the unfolding. Then, it may be possible to learn which inputs are redundant and reuse this information to discard inputs later, when SMT is too slow because of the depth of the unfolding. Furthermore, the presented analysis method may be extended to handle module sharing and active anti-tokens.

Furthermore, new approaches should be developed to leverage the fact that the performance is bounded both ways. The objective could be to reduce the total number of simulations to be performed at the end of the exploration, and thus the total run-time. Furthermore, it might be

possible to increase the diversity of the set of final solutions, providing more trade-off choices to the designer who is using the tool. For example, if two design points have very similar cycle time and throughput bounds, one of them may be discarded. This way, the final set of solutions would contain design points that in principle did not look as promising as the discarded ones. However, there would be more heterogeneity in the final pool of solutions, and at the end they might provide a more interesting pipeline.

Chapter 7

Conclusions

This thesis has presented a set of contributions on the analysis and optimization of synchronous elastic systems. This chapter gathers the conclusions derived from the previous chapters and discusses some research lines opened by this work.

7.1 Contributions

The starting point of this thesis is the novel work in elastic systems [31, 53, 82, 127], which allows to build latency-insensitive circuits within a conventional clocked synchronous framework. The synchronous elastic approach provides several technical advantages, like the ability to cut long wires without affecting the correctness of the design by adding an empty buffer.

This work shows that it is possible to perform design optimization on elastic systems by using only simple correct-by-construction local transformations. On top of this idea, an automated framework has been built to automatically explore the design space unfolded by this simple set of transformations. In particular, this work has shown that it is possible to automatically build a pipeline starting from a functional specification of the design.

A fast method to evaluate the throughput of elastic systems is essential in order to enable automatic exploration. This thesis has also presented a novel method to analyze the throughput of an elastic system, which can handle early evaluation and variable-latency units.

Automatic Exploration of Elastic Pipelines

Elasticity enables new opportunities for system optimization that target the average case performance. By using early evaluation, the designer can focus on optimizing frequently used and hence critical parts of the design. Those blocks that are not critical for the overall performance can have a relaxed timing with some extra latency cost. The obtained performance advantage can also be used for saving power.

This work has proposed a framework for automatic exploration of microarchitectures. This framework uses simple correct-by-construction transformations, presented in Chapter 3: memory bypass, retiming, multiplexor retiming, recycling, buffer resizing, introduction of early evaluation, anti-token insertion, anti-token retiming, anti-token grouping, introduction of variable-latency units and sharing of functional units. Many of these transformations are only applicable to elastic systems, since they modify the latency of computations or communications. Even though they have been proposed and used in other works, this thesis puts them together as the ground over which the rest of the contributions of this work are built.

After applying any of these transformations, the resulting system is latency equivalent to the original one, i.e., the stream of *valid* data transmitted at the outputs of the system is the same given identical input streams. This property has been verified using model checking for all transformations.

A novel method for introducing speculative execution in elastic systems is covered in Chapter 4. It is performed by applying Shannon decomposition and module sharing to a non-speculative design. Since both transformations are correct-by-construction, functional equivalence is preserved when applying speculation. It has been shown that speculation can be used to enhance performance of interesting design examples such as variable-latency units or resilient designs.

Chapter 5 shows how to build a pipeline of a closed loop system with register files and memories by applying a sequence of elastic transformations. Standard bypass and retiming alone are not sufficient to build such pipelines. This method enables designers to refine a functional model towards a cycle-accurate implementation. An automation of this process is proposed, which can effectively explore the space of possible pipelines and return a design with a near-optimal performance. It is also possible to obtain a set of pipelines with different trade-off options between performance and area.

Design space exploration is driven by certain probabilities that describe the workload expected to be run on the circuit. For a given workload, there is an optimum pipeline depth that delivers the

best possible performance. Thus, specialization of a design can significantly increase its performance. The examples in Chapter 5 show how different parameters, such as pipeline depth or data hazard probabilities, affect the performance of the system and the optimal structure of the pipeline.

The proposed exploration method relies on the features of elastic circuits. However, at the end it proposes a specific pipeline structure. All pipelines have some level of elasticity implicitly, because they must stall for data dependencies, they may allow operations with different possible latencies, etc. A designer may be interested in obtaining the best possible structure for a pipeline but may not want to pay the cost of elasticizing the whole design. It may be the case that the proposed pipeline is optimal (or near-optimal) in performance independently of whether it is implemented as a synchronous elastic circuit. The method presented in this thesis may effectively be used this way. However, only implementation using elastic systems guarantees that the performance predicted while exploring the design space will be achieved.

Performance Analysis

Chapter 6 proposes a method to analyze the performance of elastic systems with early evaluation and variable-latency units. The method computes a lower bound of the throughput by unfolding a marked graph, deriving symbolic expressions for the firing times of the transitions and evaluating them. The accuracy of the lower bound can be incremented by increasing the number of correlations taken into account when the expressions are evaluated. The worst case running time when all correlations are taken into account is exponential.

Results show that an accurate lower bound of the throughput can be obtained two orders of magnitude faster than simulation. If this is added to the known upper bound method, a good approximation of the actual throughput can be achieved quickly without running any simulations. A fast method to estimate the performance of an elastic system is important because it allows to perform microarchitectural exploration without running any simulations, which are too time consuming for the presented exploration framework.

7.2 Further Research

This section gives an overview of possible research lines that may be derived from the results of this thesis. Some enhancements to the presented method for automatic pipelining are proposed, and the extension of elastic systems in order to support multi-threading is discussed.

Improvements to the Automatic Pipelining Method

In the core of the automatic pipelining algorithm, it lies the retiming and recycling optimization procedure, which is modeled as a mixed integer linear programming problem. In this problem, every node is assigned a delay, a latency and an area. However, it would be possible to extend the model so that nodes can have various delays and latencies.

For example a node representing an adder can have different implementations: a ripple carry adder, a prefix-tree adder, etc., and variable-latency versions of these implementations with better timing but higher latency. All these choices would be available during the optimization procedure, so that the best one in timing can be chosen automatically in critical parts of the design, and an implementation with a more relaxed timing and a smaller area is chosen for parts of the design that are not critical.

Speculation may also be better embedded within the exploration framework. In this work, the exploration method can only identify places where speculative execution can fit in well structurally. However, in order to apply speculation successfully, it must be possible to predict the outcome of the computation over which speculation is being applied. Therefore, a designer must indicate which choices can be predicted and must provide a scheduler to do so.

Finally, it could be possible to improve the exploration algorithm by using the knowledge provided by the lower bound method, as discussed in the conclusions of Chapter 6. Since there is both an upper bound and a lower bound for each design point, they can be used to prune the set of interesting designs or to make it more diverse.

Multi-threading

Elastic systems implement in-order single-threaded pipelines. As a further research direction, elastic systems may be extended to support multi-threading.

There are few studies on automatic pipelining with multi-threading. In [59], a multi-threaded pipeline can be generated from a parameterized in-order pipelining template. In [122], a more general multi-threaded pipelining approach is proposed starting from an initial transactional specification.

The beauty of elastic systems is that they can be modeled using marked graphs, which allows to have formal performance analysis and optimization methods. The natural extension in order to support multi-threading is to use *colored Petri nets* [83], where each token is assigned a color (i.e., a thread). Token order must be preserved within the tokens that belong to the same color, but tokens

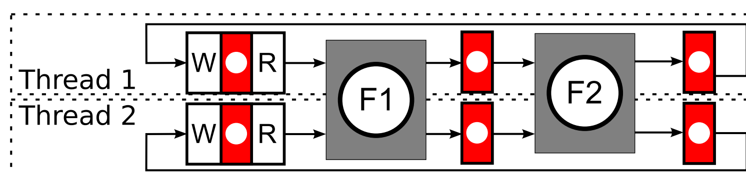


Figure 7.1: Possible implementation for elastic multi-threading support. Grey boxes are shared units.

with different colors do not have to preserve any kind of order relationship. The current analysis and optimization methods for elastic methods should be extended in order to support colors.

In the simplest approach to implement multi-threaded elastic systems, each thread would own a copy of an automatically generated pipeline. Next, the thread-independent copies may be merged using the sharing transformation. Each clock cycle, the scheduler of each shared block would decide which thread can use the block. For example, Fig. 7.1 shows a simple pipeline with two threads. Each thread has its own register file, and the functional units $F1$ and $F2$ are shared by the threads.

This approach is simple and allows total thread scheduling freedom. However, it also has a large area overhead, since each EB and the whole elastic controller is replicated once per thread. New transformations such as sharing of EBs may alleviate this overhead. A more efficient implementation would consist of adding a color identifier to the handshake protocol of elastic controllers. All the control primitives should be extended and verified accordingly. In particular, there may be different possible versions of EBs depending on whether they allow reordering of tokens with different colors (better performance), or they behave strictly as a FIFO (simpler and smaller in area).

Once a performance analysis method is developed for multi-threaded elastic systems, it may be possible to extend the optimization methods: retiming and recycling, buffer sizing and automatic pipelining methods.

Bibliography

[1] Berkeley Logic Synthesis and Verication Group. ABC: A System for Sequential Synthesis and Verification. http://www-cad.eecs.berkeley.edu/~alanmi/abc/.

[2] A. Agarwal, D. Blaauw, V. Zolotov, and S. Vrudhula. Computation and refinement of statistical bounds on circuit delay. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pages 348–353. ACM New York, NY, USA, 2003.

[3] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. *SIGARCH Computer Architecture News*, 28(2):248–259, 2000.

[4] A. Agiwal and M. Singh. An architecture and a wrapper synthesis approach for multi-clock latency-insensitive systems. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 1006–1013, 2005.

[5] M. Akian, R. Bapat, and S. Gaubert. Max-plus algebra. *Handbook of Linear algebra: Discrete Mathematics and its Application, Chapman & Hall/CRC, Baton Rouge, LA*, 2007.

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

[7] A. Armando, C. Castellini, and E. Giunchiglia. Sat-based procedures for temporal reasoning. In *ECP '99: Proc. European Conf. Planning*, pages 97–108, London, UK, 2000. Springer-Verlag.

[8] P. Ashar, S. Devadas, and A. Newton. *Sequential logic synthesis*. Springer, 1992.

[9] D. Bañeres, J. Cortadella, and M. Kishinevsky. Variable-latency design by function speculation. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1704–1709, Mar. 2009.

[10] P. Beerel, A. Lines, M. Davies, and N. Kim. Slack matching asynchronous designs. In *Proc. Int. Symp. Asynchronous Circuits and Systems*, page 11, 2006.

[11] L. Benini, E. Macii, and M. Poncino. Telescopic units: increasing the average throughput of pipelined designs by adaptive latency control. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pages 22–27, 1997.

[12] L. Benini, G. D. Micheli, A. Lioy, E. Macii, G. Odasso, and M. Poncino. Automatic synthesis of large telescopic units based on near-minimum timed supersetting. *IEEE Trans. on Computers*, 48(8):769–779, 1999.

[13] K. Bernstein, D. Frank, A. Gattiker, W. Haensch, B. Ji, S. Nassif, E. Nowak, D. Pearson, and N. Rohrer. High-performance CMOS variability in the 65-nm regime and beyond. *IBM Journal of Research and Development*, 50(4/5):449, 2006.

[14] C. Berthet, O. Coudert, and J. C. Madre. New Ideas on Symbolic Manipulations of Finite State Machines. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 224–227, 1990.

[15] E. Best. Structure theory of Petri nets: the free choice hiatus. *Petri Nets: Central Models and Their Properties*, pages 168–205, 1987.

[16] E. Bloch. The engineering design of the stretch computer. In *Proc. IRE/AIEE/ACM Eastern Joint Computer Conference*, pages 48–58, Dec. 1959.

[17] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[18] S. Borkar. Thousand core chips: a technology perspective. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pages 746–749, 2007.

[19] J. Boucaron, R. de Simone, and J. Millo. Latency-insensitive design and central repetitive scheduling. In *Proc. ACM/IEEE Int. Conf. Formal Methods and Models for Codesign (MEMOCODE)*, pages 175–183. IEEE, 2006.

[20] A. Branover, R. Kol, and R. Ginosar. Asynchronous design by conversion: converting synchronous circuits into asynchronous ones. In *Proc. Design, Automation and Test in Europe (DATE)*, volume 2, pages 870–875, 2004.

[21] C. Brej. *Early Output Logic and Anti-Tokens*. PhD thesis, University of Manchester, 2005.

[22] R. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pages 236–243, 1995.

[23] D. Bufistov. *Performance Optimization of Elastic Systems*. PhD thesis, Universitat Politecnica de Catalunya, 2010.

[24] D. Bufistov, J. Cortadella, M. Galceran-Oms, J. Júlvez, and M. Kishinevsky. Retiming and recycling for elastic systems with early evaluation. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pages 288–291, July 2009.

[25] D. Bufistov, J. Cortadella, M. Galceran-Oms, J. Júlvez, and M. Kishinevsky. Retiming and recycling for elastic systems with early evaluation. (available at http://www.lsi.upc.edu/dept/techreps/llistat_detallat.php?id=1050). Technical report, 2009.

[26] D. Bufistov, J. Cortadella, M. Kishinevsky, and S. Sapatnekar. A general model for performance optimization of sequential systems. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 362–369, 2007.

[27] D. Bufistov, J. Júlvez, and J. Cortadella. Performance optimization of elastic systems using buffer resizing and buffer insertion. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 442–448, Nov. 2008.

[28] S. Burns. *Performance analysis and optimization of asynchronous circuits*. PhD thesis, California Institute of Technology, 1990.

[29] J. Campos, G. Chiola, J. Colom, and M. Silva. Tight polynomial bounds for steady-state performance of marked graphs. In *Workshop on Petri Nets and Performance Models, (PNPM)*, pages 200–209, 1989.

[30] J. Campos, G. Chiola, and M. Silva. Ergodicity and throughput bounds of Petri nets with unique consistent firing count vector. *IEEE Trans. on Software Engineering*, 17(2):117–125, 1991.

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

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

[33] L. Carloni and A. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, page 367, 2000.

[34] L. Carloni and A. Sangiovanni-Vincentelli. Coping with latency in SoC design. *IEEE Micro, Special Issue on Systems on Chip*, 22(5):12, October 2002.

[35] 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.

[36] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin. Elastic circuits. *IEEE Trans. on Computer-Aided Design*, 28(10):1437–1455, Oct. 2009.

[37] J. Carmona, J. Júlvez, J. Cortadella, and M. Kishinevsky. Scheduling Synchronous Elastic Designs. In *Conf. Application of Concurrency to System Design (ACSD)*, pages 52–59. IEEE, 2009.

[38] M. Casu. Improving synchronous elastic circuits: Token cages and half-buffer retiming. In *Proc. Int. Symp. Asynchronous Circuits and Systems*, pages 128–137, 2010.

[39] M. Casu and L. Macchiarulo. A new approach to latency insensitive design. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pages 576–581, June 2004.

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

[41] M. Casu and L. Macchiarulo. Adaptive Latency Insensitive Protocols and Elastic Circuits with Early Evaluation: A Comparative Analysis. *Electronic Notes in Theoretical Computer Science*, 245:35–50, 2009.

[42] S. Chakraborty, K. Yun, and D. Dill. Timing analysis of asynchronous systems using time separation of events. *IEEE transactions on computer-aided design of integrated circuits and systems*, 18(8):1061–1076, 1999.

[43] T. Chelcea and S. Nowick. Robust interfaces for mixed-timing systems. *IEEE Trans. VLSI Syst.*, 12(8):857–873, 2004.

[44] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. *Lecture Notes in Computer Science*, pages 359–364, 2002.

[45] R. Collins and L. P. Carloni. Topology-based optimization of maximal sustainable throughput in a latency-insensitive system. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pages 410–416, June 2007.

[46] M. Comeau and K. Thulasiraman. Structure of the Submarking-Reachability Problem and Network Programming. *IEEE Trans. on Circuits and Systems*, 35(1), 1988.

[47] J. Cong. Challenges and opportunities for design innovations in nanometer technologies. *Invited Semiconductor Research Corporation Design Sciences Concept Paper*, pages 1–15, may 1998.

[48] J. Cortadella, M. Galceran-Oms, and M. Kishinevsky. Elastic systems. In *Proc. ACM/IEEE Int. Conf. Formal Methods and Models for Codesign (MEMOCODE)*, pages 149–158, July 2010.

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

[50] J. Cortadella and M. Kishinevsky. Synchronous elastic circuits with early evaluation and token counterflow. (available at http://www.lsi.upc.edu/dept/techreps/ llistat_detallat.php?id=958). Technical report, 2007.

[51] J. Cortadella, M. Kishinevsky, D. Bufistov, J. Carmona, and J. Júlvez. Elasticity and Petri Nets. *Trans. on Petri Nets and Other Models of Concurrency I*, 5100:221–249, Aug. 2008.

[52] J. Cortadella, M. Kishinevsky, and B. Grundmann. SELF: Specification and design of a synchronous elastic architecture for DSM systems. In *TAU-2006: Int. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 2006.

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

[54] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou. Desynchronization: Synthesis of asynchronous circuits from synchronous specifications. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 25(10):1904–1921, 2006.

[55] I. CPLEX. 10.0 user's manual. *ILOG SA, Gentilly, France*, 2005.

[56] G. Dantzig. Linear Programming and Extensions. 1963.

[57] A. Dasdan, S. Irani, and R. Gupta. Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, page 42. ACM, 1999.

[58] G. De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[59] R. Dimond, O. Mencer, and W. Luk. Application-specific customisation of multi-threaded soft processors. In *IEEE Proc. Computers and Digital Techniques*, volume 153, pages 173–180. IET, 2006.

[60] B. Dutertre and L. De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2006.

[61] J. Esparza and K. Heljanko. *Unfoldings: a partial-order approach to model checking*. Springer-Verlag, 2008.

[62] S. Eyerman, L. Eeckhout, and K. De Bosschere. Efficient design space exploration of high performance embedded out-of-order processors. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 351–356, 2006.

[63] M. Galceran-Oms, J. Cortadella, and M. Kishinevsky. Symbolic performance analysis of elastic systems. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 778–785. IEEE.

[64] M. Galceran-Oms, J. Cortadella, and M. Kishinevsky. Speculation in elastic systems. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pages 292–295, July 2009.

[65] M. Galceran-Oms, J. Cortadella, M. Kishinevsky, and D. Bufistov. Automatic microarchitectural pipelining. In *Proc. Int. Workshop on Logic Synthesis*, pages 214–221, June 2009.

[66] M. Galceran-Oms, J. Cortadella, M. Kishinevsky, and D. Bufistov. Automatic microarchitectural pipelining. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 961–964, Mar. 2010.

[67] G. Gill, V. Gupta, and M. Singh. Performance estimation and slack matching for pipelined asynchronous architectures with choice. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 449–456, 2008.

[68] A. Gotmanov, M. Kishinevsky, and M. Galceran-Oms. Evaluation of flexible latencies: designing synchronous elastic H.264 CABAC decoder. In *In Proc. of the Problems in design of micro- and nano-electronic systems (in Russian)*, Oct. 2010.

[69] P. L. Guernic, J.-P. Talpin, and J.-C. L. Lann. Polychrony for system design. *Journal of Circuits, Systems and Computers*, 12(3):261–304, Apr. 2003.

[70] J. Gunawardena. Timing analysis of digital circuits and the theory of min-max functions. In *TAU'93, ACM Int. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1993.

[71] G. Hachtel and F. Somenzi. *Logic synthesis and verification algorithms*. Kluwer Academic Pub, 1996.

[72] A. Hartstein and T. R. Puzak. The optimum pipeline depth for a microprocessor. In *Proc. Int. Symp. Computer Architecture*, pages 7–13, 2002.

[73] S. Hassoun and C. Ebeling. Architectural retiming: Pipelining latency-constrained circuits. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pages 708–713, June 1996.

[74] S. Hassoun and C. Ebeling. Using precomputation in architecture and logic resynthesis. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 316–323, 1998.

[75] J. Hennessy and D. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann Publisher Inc., 1990.

[76] J. Higgins and M. Aagaard. Simplifying the design and automating the verification of pipelines with structural hazards. *ACM Trans. Design Automation of Electronic Systems (TODAES)*, 10(4):672, 2005.

[77] J. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 511–519. IEEE Press Piscataway, NJ, USA, 2000.

[78] G. Hoover and F. Brewer. Synthesizing synchronous elastic flow networks. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 306–311. ACM, 2008.

[79] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. *ACM SIGARCH Computer Arch.*, 30(2):14–24, 2002.

[80] H. Hulgaard, S. Burns, T. Amon, and G. Borriello. An algorithm for exact bounds on the time separation of events in concurrent systems. *IEEE transactions on computers*, 44(11):1306–1317, 1995.

[81] A. P. Hurst, A. Mishchenko, and R. K. Brayton. Scalable min-register retiming under timing and initializability constraints. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pages 534–539, June 2008.

[82] 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. Int. Symp. Asynchronous Circuits and Systems*, pages 3–12, Apr. 2002.

[83] K. Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use*. Springer-Verlag, 1997.

[84] A. Jerraya. Long term trends for embedded system design. In *Proc. CEPA 2 Workshop*, 2005.

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

[86] J. Júlvez, J. Cortadella, and M. Kishinevsky. On the performance evaluation of multi-guarded marked graphs with single-server semantics. *Discrete Event Dynamic Systems*, 20(3):377–407, Sept. 2010.

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

[88] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–396, 1984.

[89] R. Karp. A characterization of the minimum mean-cycle in a digraph. *Discrete Maths*, 23:309–311, 1978.

[90] L. Khachiyan. A polynomial algorithm for linear programming (Russian). In *Doklady Akademii Nauk SSSR 244*, pages 1093–1096, 1979.

[91] E. Kilada and K. Stevens. Control Network Generator For Latency Insensitive Designs. In *Proc. Design, Automation and Test in Europe (DATE)*, Apr. 2010.

[92] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent hardware: the theory and practice of self-timed design*. John Wiley & Sons, Inc., 1994.

[93] R. Kol and R. Ginosar. A doubly-latched asynchronous pipeline. In *Proc. IEEE Int. Conf. Computer Design: VLSI in Computers and Processors (ICCD)*, pages 706–711, 1997.

[94] Y. Kondo, N. Ikumi, K. Ueno, J. Mori, and M. Hirano. An early-completion-detecting ALU for a 1 GHz 64b datapath. In *IEEE Int. Conf. Solid-State Circuits (ISSCC)*, pages 418–419, 1997.

[95] D. Kroening and W. Paul. Automated pipeline design. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pages 810–815, 2001.

[96] S. Krstić, J. Cortadella, M. Kishinevsky, and J. O'Leary. Synchronous elastic networks. In *Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, Nov. 2006.

[97] P. Kudva, G. Gopalakrishnan, E. Brunvand, and V. Akella. Performance Analysis and Optimization of Asynchronous Circuits. In *Proc. IEEE Int. Conf. Computer Design: VLSI in Computer & Processors*, pages 221–224, 1994.

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

[99] C.-H. Li and L. Carloni. Using functional independence conditions to optimize the performance of latency-insensitive systems. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2007.

[100] C.-H. Li, R. Collins, S. Sonalkar, and L. P. Carloni. Design, implementation, and validation of a new class of interface circuits for latency-insensitive design. In *Proc. ACM/IEEE Int. Conf. Formal Methods and Models for Codesign (MEMOCODE)*, pages 13–22, 2007.

[101] J. Liou, K. Cheng, S. Kundu, and A. Krstic. Fast statistical timing analysis by probabilistic event propagation. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pages 661–666, 2001.

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

[103] R. Lu and C.-K. Koh. Performance optimization of latency insensitive systems through buffer queue sizing of communication channels. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 227–231, Nov. 2003.

[104] R. Manohar and A. Martin. Slack elasticity in concurrent computing. In *Mathematics of Program Construction*, pages 272–285. Springer, 1998.

[105] M.-C. V. Marinescu and M. C. Rinard. High-level automatic pipelining for sequential circuits. In *Proc. Int. Symp. on Systems Synthesis*, pages 215–220, 2001.

[106] J. Matthews and J. Launchbury. Elementary microarchitecture algebra. *Lecture Notes in Computer Science*, 1633:288–300, 1999.

[107] K. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Computer Aided Verification*, pages 164–177, 1992.

[108] K. McMillan. Verification of infinite state systems by compositional model checking. *Correct Hardware Design and Verification Methods*, 1703:219–237, 1999.

[109] K. L. McMillan. A compositional rule for hardware design refinement. In *Computer Aided Verification*, pages 24–35. Springer, 1997.

[110] E. Mercer and C. Myers. Stochastic cycle period analysis in timed circuits. In *IEEE Int. Symp. Circuits and Systems (ISCAS)*, volume 2, pages 172–175, 2000.

[111] P. Mishra, A. Kejariwal, and N. Dutt. Synthesis-driven exploration of pipelined embedded processors. In *Proc. Int. Conf. VLSI Design*, pages 921–926, 2004.

[112] D. Misunas. Petri nets and speed independent design. *Communications of the ACM*, 16(8):481, 1973.

[113] J. Monteiro, S. Devadas, and A. Ghosh. Retiming sequential circuits for low power. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 398–402, 1993.

[114] G. Moore. Cramming more components onto integrated circuits. 38(8):114–117, Apr. 1965.

[115] T. Murata. Petri Nets: Properties, analysis and applications. *Proc. of the IEEE*, pages 541–580, Apr. 1989.

[116] K. G. Murty. *Linear Programming*. Wiley and Sons, 1983.

[117] S. R. Nassif. Design for variability in dsm technologies. In *Proc. Int. Symp. Quality of Electronic Design (ISQED)*, pages 451–454, 2000.

[118] C. Nielsen and M. Kishinevsky. Performance analysis based on timing simulation. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pages 70–76, 1994.

[119] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13(1):85–108, 1981.

[120] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T). *Journal of the ACM (JACM)*, 53(6):977, 2006.

[121] E. Nurvitadhi, J. Hoe, T. Kam, and S. Lu. Automatic Pipelining from Transactional Datapath Specifications. *Proc. Design, Automation and Test in Europe (DATE)*, pages 1001–1004, Mar. 2010.

[122] E. Nurvitadhi, J. Hoe, S. Lu, and T. Kam. Automatic multithreaded pipeline synthesis from transactional datapath specifications. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pages 314–319, 2010.

[123] J. O'Leary and G. Brown. Synchronous emulation of asynchronous circuits. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 16(2):205–209, 2002.

[124] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[125] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1995.

[126] N. Park and A. Parker. Sehwa: a software package for synthesis of pipelines frombehavioral specifications. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 7(3):356–370, 1988.

[127] A. Peeters and K. Van Berkel. Synchronous handshake circuits. In *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, pages 86–95, March 2001.

[128] A. Pnueli. The temporal logic of programs. In *Proc. IEEE Symp. Foundations of Computer Science*, pages 46–57, 1977.

[129] M. D. Powell, A. Biswas, J. S. Emer, S. S. Mukherjee, B. R. Sheikh, and S. Yardi. CAMP: A technique to estimate per-structure power at run-time using a few simple parameters. In *IEEE Int. Symp. High Performance Computer Architecture (HPCA)*, pages 289–300, Feb. 2009.

[130] P. Prakash and A. Martin. Slack matching quasi delay-insensitive circuits. In *Proc. Int. Symp. Asynchronous Circuits and Systems*, page 10, 2006.

[131] C. Ramamoorthy and G. Ho. Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets. *IEEE Trans. on Software Engineering*, 6(5):440–449, 1980.

[132] C. Ramchandani. Analysis of asynchronous concurrent systems by timed petri nets. Technical report, Project MAC, TR-120, MIT, 1974.

[133] R. Reese, M. Thornton, and C. Traver. A coarse-grain phased logic cpu. *IEEE Trans. Comput.*, 54:788–799, July 2005.

[134] R. Reese, M. Thornton, C. Traver, and D. Hemmendinger. Early evaluation for performance enhancement in phased logic. *IEEE Trans. Computer-Aided Design*, 24(4):532–550, Apr. 2005.

[135] J. Rodriguez-Beltran and A. Ramfrez-Trevino. Minimum initial marking in timed marked graphs. In *IEEE Int. Conf. Systems, Man, and Cybernetics*, volume 4, pages 3004–3008, 2000.

[136] S. S. Sapatnekar. *Timing*. Kluwer Academic Publishers, 2004.

[137] S. S. Sapatnekar and R. B. Deokar. Utilizing the timing skew equivalence in a practical algorithm for retiming large circuits. *IEEE Trans. Computer-Aided Design*, 15(10):1237–1248, Oct. 1996.

[138] P. Saxena, N. Menezes, P. Cocchini, and D. Kirkpatrick. Repeater scaling and its impact on CAD. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 23(4):451–463, 2004.

[139] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis University of California at Berkeley. *UCB/ERL M*, 1992.

[140] J. Shen and M. Lipasti. *Modern processor design: fundamentals of superscalar processors*. McGraw-Hill Science Engineering, 2004.

[141] N. V. Shenoy. Retiming: Theory and practice. *Integration, the VLSI Journal*, 22(1):1–21, 1997.

[142] M. Silva. Introducing petri nets. *Practice of Petri Nets in Manufacturing*, pages 1–62, 1993.

[143] M. Singh and M. Theobald. Generalized latency-insensitive systems for single-clock and multi-clock architectures. In *Proc. Design, Automation and Test in Europe (DATE)*, page 21008, 2004.

[144] C. Soviani, O. Tardieu, and S. Edwards. Optimizing sequential cycles through Shannon decomposition and retiming. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1085–1090, 2006.

[145] L. Spracklen and S. Abraham. Chip multithreading: Opportunities and challenges. In *Proc. Int. Symp. High-Performance Computer Architecture*, pages 248–252, 2005.

[146] S. Srinivasan, K. Sarker, and R. Katti. Token-Aware Completion Functions for Elastic Processor Verification. *Research Letters in Electronics*, 2009.

[147] Y.-S. Su, D.-C. Wang, S.-C. Chang, and M. Marek-Sadowska. An efficient mechanism for performance optimization of variable-latency designs. In *Proc. ACM/IEEE Design Automation Conf. (DAC)*, pages 976–981, 2007.

[148] S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla. Validating families of latency insensitive protocols. *IEEE Trans. Comput.*, 55:1391–1401, November 2006.

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

[150] H. J. Touati and R. K. Brayton. Computing the initial states of retimed circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 12(1):157–162, 1993.

[151] V. Varshavsky and V. Marakhovsky. GALA (Globally Asynchronous: Locally Arbitrary) Design. *Concurrency and Hardware Design*, pages 61–107, 2002.

[152] M. Vijayaraghavan and Arvind. Bounded dataflow networks and latency-insensitive circuits. In *Proc. ACM/IEEE Int. Conf. Formal Methods and Models for Codesign (MEMOCODE)*, July 2009.

[153] J. Wakerly, C. Jong, and C. Chang. *Digital Design: Principles and Practices*. Prentice Hall Englewood Cliffs, NJ, 2001.

[154] T. Williams. Performance of iterative computation in self-timed rings. *The Journal of VLSI Signal Processing*, 7(1):17–31, 1994.

[155] R. W. Wolff. *Stochastic modeling and the theory of queues*. Prentice Hall, 1989.

[156] A. Xie and P. Beerel. Symbolic techniques for performance analysis of timed systems based on average time separation of events. In *Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, pages 64–75, 1997.

[157] A. Xie and P. Beerel. Performance analysis of asynchronous circuits and systems using stochastic timed Petri nets. In *Hardware Design and Petri Nets*, 1999.

[158] Xilinx. Single Error Correction and Double Error Detection (SECDED) with CoolRunner-II CPLDs. *Application Note XAPP383*, 1:1–4, 2003.

[159] 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*, 9:189–233, 1996.

Index

SELF protocol, 21, 30

adaptive latency-insensitive designs, 28
anti-tokens, 28, 63, 64, 75, 94, 95, 114, 125
asynchronous designs, 5

backpressure, 19, 21, 58, 125
bubble insertion, 57, 74, 94
buffer capacity, 58
bypass, 54, 88, 95

correct-by-construction, 10, 49, 69, 73, 89
critical core, 61, 113
cycle time, 2, 21

data hazard, 92, 96
data variability, 61
deadlock, 58, 69

early evaluation, 28, 54, 60, 73, 75, 78, 88, 94, 119, 124, 130
effective cycle time, 21
elastic buffers, 19, 22, 31, 69, 78
elastic channel, 19
elastic marked graph, 41
elastic module, 18
error correction, 80

fork, 26, 33
functional independence conditions, 28

granularity, 37

infinite-server semantics, 16, 104, 127
integer linear programming, 17, 59, 94, 106

join, 26, 33

latency-insensitive designs, 18
linear programming, 17, 45
little care, 61, 113

MAREX, 53, 102
marked graph, 16, 41, 117, 124
Markov chain, 45, 89
microarchitectural graph, 51, 94
mixed integer linear programming, 59
module sharing, 67, 69, 77
multiplexor, 54
multiplexor retiming, 57

NuSMV, 53

Petri net, 15, 41, 126
pipeline, 3
pipelining, 11, 46, 87, 88, 92, 99, 103, 109

recycling, 57, 94
register file, 40, 52
retiming, 56, 94
retiming and recycling, 58, 94

scheduling, 37
single-server semantics, 16, 125, 127
skid-buffer, 25, 58
slack matching, 58
speculation, 11, 73
synchronous designs, 2
synthesis flow, 39, 50

throughput, 17, 20, 127, 151
timed marked graph, 16, 41, 124
toolkit, 53
transfer equivalence, 20, 69
transformations, 10, 49

unfolding, 126, 146

variable-latency, 52, 66, 82, 118, 125, 130
verification, 35, 69
Verilog, 53
video decoder, 62, 113